28th International Conference on Types for Proofs and Programs

TYPES 2022, June 20–25, 2022, LS2N, University of Nantes, France

Edited by

Delia Kesner
Pierre-Marie Pédrot

Editors

Delia Kesner
Université Paris Cité, France
kesner@irif.fr

Pierre-Marie Pédrot
INRIA and LS2N, Nantes, France
pierre-marie.pedrot@inria.fr

ACM Classification 2012
Theory of computation → Type theory; Theory of computation → Type structures; Computing methodologies → Representation of mathematical objects; Theory of computation → Interactive proof systems; Theory of computation → Logic; Theory of computation → Logic and verification; Theory of computation → Proof theory; Theory of computation → Constructive mathematics; Theory of computation → Linear logic; Theory of computation → Process calculi; Software and its engineering → Formal software verification; Security and privacy → Systems security

ISBN 978-3-95977-285-3

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-285-3.

Publication date
August, 2023

Bibliographic information published by the Deutsche Nationalbibliothek
The Deutsche Nationalbibliothek lists this publication in the Deutsche Nationalbibliografie; detailed bibliographic data are available in the Internet at https://portal.dnb.de.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2022.0

LIPIcs – Leibniz International Proceedings in Informatics

LIPIcs is a series of high-quality conference proceedings across all fields in informatics. LIPIcs volumes are published according to the principle of Open Access, i.e., they are available online and free of charge.

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

Papers

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Preface

The TYPES meetings are a forum to present new and ongoing work in all aspects of type theory and its applications, especially in formalised and computer assisted reasoning and computer programming. This volume constitutes the post-proceedings of the 28th International Conference on Types for Proofs and Programs, TYPES 2022, that was held in LS2N, University of Nantes, from 20 to 25 June 2022.

The meetings from 1990 to 2008 were annual workshops corresponding to five consecutive EU-funded networking projects. Since 2009, TYPES has been run as an independent conference series. Previous TYPES meetings were organised by Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), Novi Sad (2016), Budapest (2017), Braga (2018), Oslo (2019), Turin (2020), Leiden (2021). The two last meetings were virtual, because of the SARSCoV-2 pandemics.

The TYPES areas of interest include, but are not limited to: Foundations of type theory and constructive mathematics; Homotopy type theory; Applications of type theory; Dependently typed programming; Industrial uses of type theory technology; Meta-theoretic studies of type systems; Proof assistants and proof technology; Automation in computer-assisted reasoning; Links between type theory and functional programming; Formalizing mathematics using type theory; Type theory in linguistics.

The TYPES conferences are all based on contributed talks based on short abstracts; reporting work in progress and work presented or published elsewhere. A post-proceedings volume is prepared after the conference, whose papers must represent unpublished work. Submitted papers to the post-proceedings are subject to a full peer-review process.

The conference programme of TYPES 22 consisted of 14 long contributed talks (20 minutes), 57 short contributed talks (10 min), and four invited talks (one hour) by Youyou Cong (Tokyo Institute of Technology), Ekaterina Komendantskaya (Heriot-Watt University), Sam Lindley (University of Edinburgh) and Leonardo de Moura (Microsoft Research). The conference was a successful event with 138 registered participants. All the details of the conference can be found at `https://types22.inria.fr`.

Concerning the post-proceedings, 18 papers were initially submitted, out of which 15 were accepted. We thank all the authors and reviewers for their hard work to make this possible! Finally, we would like to thank CNRS, Inria, and the COST Action CA20111 for sponsoring the conference, and Nantes Université and LS2N for kindly covering the costs of the post-proceedings.

Delia Kesner and Pierre-Marie Pédrot, June 2023.

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot
LIPICS Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

All Watched Over by Machines of Loving Grace

Dominic P. Mulligan ✉ 🏠

Automated Reasoning Group, Amazon Web Services, Cambridge, UK[1]

Abstract

Modern operating systems are typically built around a trusted system component called the *kernel* which amongst other things is charged with enforcing system-wide security policies. Crucially, this component must be kept isolated from untrusted software at all times, which is facilitated by exploiting machine-oriented notions of separation: private memories, privilege levels, and similar.

Modern proof-assistants are typically built around a trusted system component called the *kernel* which is charged with enforcing system-wide soundness. Crucially, this component must be kept isolated from untrusted automation at all times, which is facilitated by exploiting programming-language notions of separation: module-private data structures, type-abstraction, and similar.

Whilst markedly different in purpose, in some essential ways operating system and proof-assistant kernels are tasked with the same job, namely enforcing system-wide invariants in the face of unbridled interaction with untrusted code. Yet the mechanisms through which the two types of kernel protect themselves are significantly different.

In this paper, we introduce *Supervisionary*, the kernel of a programmable proof-checking system for Gordon's HOL, organised in a manner more reminiscent of an operating system than a typical LCF-style proof-checker. Supervisionary's kernel executes at a relative level of privilege compared to untrusted automation, with trusted and untrusted system components communicating across a limited system call boundary. Kernel objects, managed on behalf of user-space by Supervisionary, are referenced by handles and are passed back-and-forth by system calls. Unusually, Supervisionary has no "metalanguage" in the LCF sense, as the language used to implement the kernel, and the language used to implement automation, need not be the same. *Any* programming language can be used to implement automation for Supervisionary, providing the resulting binary respects the kernel calling convention and binary interface, with no risk to system soundness. Lastly, Supervisionary allows arbitrary programming languages to be endowed with facilities for proof-checking. Indeed, the handles that Supervisionary uses to denote kernel objects may be thought of as an extremely expressive form of *capability* – in the computer security sense of that word – and can potentially be used to enforce fine-grained correctness and security properties of programs at runtime.

2012 ACM Subject Classification Theory of computation → Higher order logic; Theory of computation → Automated reasoning; Theory of computation → Logic and verification; Software and its engineering → Operating systems

Keywords and phrases Proof assistant design, operating systems, HOL, LCF, Supervisionary, system description, capabilities

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.1

Supplementary Material *Software (Source Code)*: `https://github.com/DominicPM/supervisionary`
archived at `swh:1:dir:7478757cd08c06735cf3a1a056246d0200100c45`

Acknowledgements We would like to thank Nick Spinale for many insightful conversations regarding Supervisionary, and Nathan Chong and two anonymous referees for their helpful feedback on earlier drafts of this paper.

[1] All work done whilst employed within the Systems Research Group, Arm Research, Cambridge.

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 1; pp. 1:1–1:23
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

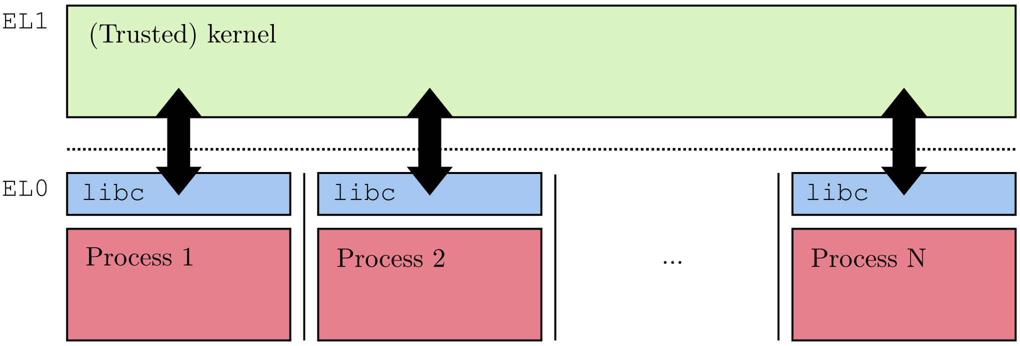


Figure 1 A schematic of the typical system organization of a commodity operating system and its associated user-space. The kernel (in green) executes at a relative level of privilege, enforced by hardware, compared to processes executing in user-space (red) – we follow the Arm convention and show the kernel executing at `EL1` and user-space at `EL0`. The two communicate across a system call boundary (dashed line) using system calls (black arrows). User-space programs are typically written making use of an abstraction library, such as `libc` (blue), to abstract over this kernel interface.

1 Introduction

This paper studies the intersection of operating system design and implementations of the foundations of mathematics. Research into the confluence of these two topics is, admittedly, a rather moribund affair at the moment. Nevertheless, with this paper we hope to convince the reader that probing the intersection of these two areas is potentially very interesting by introducing *Supervisionary*, a programmable proof-checking system for Gordon's HOL. This system has a novel system design, with some interesting properties, and moreover some interesting consequences. We first, however, begin with a scene-setting overview of common principles in operating system design and implementation.

1.1 On operating systems

Most commodity operating systems – that is, Microsoft Windows and Unix-derivatives[2] – fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel* [35].

The kernel is the sole component that can interface unfettered with all system resources, including devices and other system hardware. Untrusted user-space applications make use of kernel interfaces in order to make use of a device or any other system resource managed by the kernel. As a result, the kernel is essentially a "pinch point" for gating access to system resources. The kernel also introduces a process abstraction in user-space and is responsible for ensuring the confidentiality and integrity of concurrently-executing processes, each of which are mutually mistrusting. The kernel is therefore *the* key component responsible for enforcing system-wide security policies, and essentially forms the "root of all trust" within a computing system. It is therefore imperative that the kernel is itself isolated sufficiently from user-space software at all times, lest this role be undermined by a malefactor.

The kernel self-isolates by co-operating with its host hardware. In support of this, mainstream microprocessors have, over the years, accreted a variety of now-familiar security features that an operating system kernel can use to defend itself from prying or interference. These include *exception levels* or *privilege rings*, as they are variously called, depending on the instruction set architecture, and which introduce a notion of *privilege* into the system.

[2] *Commodity* here is used to guard against pedantic quibbling over research operating system designs – like exokernels [9] and other oddities – which arguably do not fit this pattern.

Here, software executing at higher-privilege – in our case, an operating system kernel[3] – gains permission to program sensitive system registers, adjust hardware operating frequencies and voltages, and generally control how the system operates. Moreover, software executing at a higher-level of privilege can "peer in" and potentially modify the runtime state of software executing at a relatively lower-level of privilege, for example reading data from, or writing data to, a buffer within the memory space of an untrusted user-space process.

Modern microprocessors also provide a form of memory management built around page tables (see e.g. [3]). These data structures have a dual role: primarily, they are used for the virtualisation of system memory via address translation, granting user-space software the illusion that it owns the entire physical address space of the machine, presenting a *virtual* address space to user-space programs. This translation process induces a notion of ownership of pages of physical memory within the system, with a page of physical memory "owned" by a principal – either the operating system, a user-space process, or both – if it is *mapped in* to that principal's address space. Moreover, page tables are also used for storing the metadata attributes of pages of memory, including read-write-execute permissions. By correctly initialising and managing these tables the kernel can keep its own code and data structures isolated – in a kernel-private memory area – that only it can access, safe from prying or interference by untrusted user-space. As a result, for systems software on modern computers, isolation is enforced by a mix of low-level machine mechanisms: separate address spaces, private memory regions, and machine-enforced privilege checks on executing software.

To make itself useful, the kernel exposes a limited interface, used by user-space to request intercession by the kernel on its behalf – for example by granting user-space access to some device, the filesystem, a socket, or some other system resource under kernel management. Dealing in generalities, to do this, the kernel exposes a suite of largely synchronous *system calls* which can be invoked by user-space programs with dedicated machine instructions provided by the microprocessor – see Figure 1 for a diagrammatic schematic, for example. On Arm platforms, with which the author is most familiar, these instructions induce a processor exception, forcing a *context switch* which flips the flow of control into the kernel's system call handler, before eventually returning the flow of control back to the calling user-space program. From user-space's point-of-view, system calls therefore have the appearance and effect of very CISC-like machine instructions, with the operating system kernel essentially presenting itself to user-space as *silicon by other means*, extending the user-space fragment of the instruction set architecture of the microprocessor with new macro instructions.

Note that for this two-way dance to work, user-space and the kernel must work together by adopting a series of joint conventions. These include a *calling convention* describing how arguments and results are passed back-and-forth across the system call interface, and a *binary interface* detailing how system calls are identified, how errors are reported back to user-space, and other miscellanea.[4] To help programmers adhere to these conventions, the operating system typically provides an abstraction layer to user-space, which on Unix variants typically takes the form of the system's C library, `libc`. Generally, this is just a convenience, and user-space software may invoke system calls directly if wanted by invoking the correct machine instruction and adhering to the appropriate calling convention.[5]

[3] Note that *Cloud hosting* as a viable business proposition essentially rests on this trick being repeated again, with a hypervisor sat in a position of privilege compared to an operating system kernel – executing out of an even higher exception level – and enforcing separation betwixt operating system instances.

[4] For more detail on the role of the system ABI, its other aspects, and its very real effects on the semantics of executing programs, see this [18] outrageously well-written yet criminally under-cited overview.

[5] This is the case on Linux, though does not hold universally on all Unix derivatives. For example Apple's MacOS and some BSD Unix variants generally consider the programming interface of the system C library as the interface of the kernel, proper, in some cases preventing any user-space code other than the system's `libc` library from invoking system calls directly, as a security mechanism.

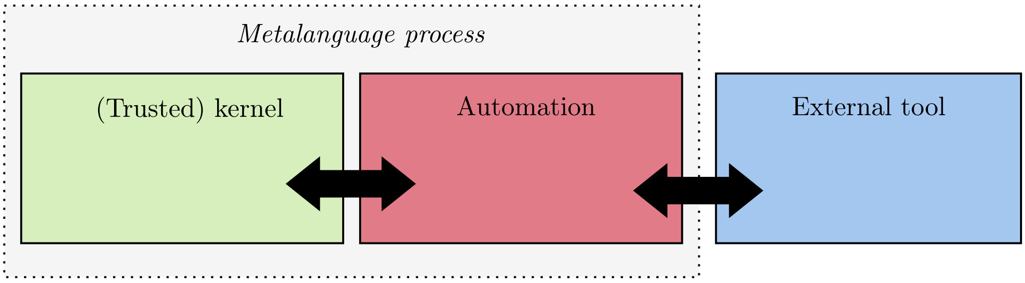


Figure 2 A schematic of the system organisation of a typical LCF-style proof assistant. The trusted kernel (green) is linked against untrusted automation (red) existing within the same metalanguage process (dotted line) and communicate with each other using the kernel's API (leftmost black arrow). External tools existing as separate processes (blue), must communicate with a shim layer written in the proof assistant's metalanguage to access the kernel (rightmost black arrow).

However, crucially, it is *generally* not the case that the operating system kernel and untrusted user-space applications must be written in the same programming language for this all to work. Whilst most operating system kernels are written in C, or a C-language derivative, user-space programs can be written in a variety of languages, and are also commonly composed of multiple libraries, written in different programming languages, linked together. Despite this, all are able to make use of system resources exposed by the kernel's system call interface by ensuring that they adhere to the calling convention and binary interface expected by the kernel. In this respect, for commodity operating systems, the C-language may have prominence as a favoured language of system implementation, but by-and-large it is not *special* or given an unduly prominent status by the kernel itself.

1.2 On programmable proof-checkers

Most modern proof-assistants – for example, systems in the wider HOL family [27, 14, 33], Coq [15], Matita [4], NuPRL [2], and similar – fit a common pattern and are architected around a relatively self-contained, trusted component typically called the system *kernel*.

The system kernel is the sole component that can authenticate claims as legitimate theorems of the implemented logic. Untrusted automation, residing outside of the kernel, must "drive" the kernel to derive a theorem on its behalf. The kernel is therefore *the* component responsible for ensuring system-wide soundness, and represents the "root of all trust" within the system. It is therefore imperative that the kernel is able to isolate itself sufficiently from untrusted automation at all times. This kernel-centric method of system organisation is known as *the LCF approach* after Milner's eponymous system [11]. See Figure 2 for a diagrammatic representation.

Most modern proof-assistants tend to be written in a *metalanguage* which serves as the implementation language for both the kernel and the majority of the untrusted automation that modern proof-assistants provide to users. This metalanguage is typically a strongly-typed functional programming language, for example an ML derivative such as OCaml or SML [22], and which offers strong modularity and abstraction features. The kernel exploits these programming language features to hide its own data structures from untrusted automation and expose a carefully limited API for proof-construction and manipulation. Notably, in an LCF-style system, the *only* mechanism automation has for constructing an authenticated theorem is by using this API, with the inference rules of the logic exposed as a suite of *smart constructors* manipulating an abstract type of theorems. The kernel is therefore a "pinch point" for any proof-construction activity within the system.

Untrusted automation and the system kernel are linked together, and reside side-by-side in the same process when the proof-assistant is executed. As a result, system soundness ultimately rests on the soundness of the implementation metalanguage's type-system – specifically its ability to correctly isolate module-private data structures and enforce type abstraction. Moreover, for systems that use ephemeral proof construction, and lack an explicit notion of serialised proof-representation such as a *proof-term* or similar, the system metalanguage is unique amongst all programming languages in that it is the *only* language capable of interfacing directly with the kernel which is, after all, "just" a module written written in that language like any other. Whilst an external tool, or automation written in another programming language, *can* interface with the kernel, it must do so indirectly, making use of a shim layer written in the system metalanguage.

1.3 Introducing the Supervisionary system

As the text above intimates, the role of the kernel in both an operating system and in a proof-assistant is – at least in an abstract sense – the same: both components must enforce system-wide invariants in the face of unbridled interaction with untrusted code; both components also act as the "root of all trust" for their respective systems; both components act as "pinch points" that untrusted code cannot help interact with if it wishes to engage in some kernel-gated activity. Consequently, both types of kernel need to correctly isolate their data structures and runtime state from interference by untrusted code. However, the two mechanisms through which this self-isolation is enforced are different: for operating system kernels[6] self-isolation is enforced using machine-oriented mechanisms; for LCF-style proof-assistants, self-isolation is enforced using programming language-oriented mechanisms.

In this paper we introduce *Supervisionary*, the kernel of a novel programmable proof-assistant for Gordon's HOL.[7] Supervisionary's design has more in common with a typical operating system than comparable implementations of HOL. Specifically, the Supervisionary kernel executes at a relative level of privilege compared to untrusted automation, which can be thought of as executing as a process in something akin to Supervisionary's version of "user-space". The trusted kernel, and untrusted user-space, communicate across a system call boundary which is carefully designed in order to maintain system soundness.

One consequence of this design is that the Supervisionary kernel immediately takes on a different character to an LCF kernel. All of the paraphernalia of a typical HOL implementation – type-formers, types, constants, terms, and theorems – are managed as *kernel objects* kept safely under the management of the kernel itself, in kernel-private memory areas. These kernel objects are never exposed *directly* to user-space, rather, they are manipulated by the Supervisionary kernel on user-space's behalf. Handles – which can be thought of as pointers, indexing Supervisionary's private memories – are used by a user-space process to identify kernel objects that the kernel should manipulate or query.

Notably, Supervisionary is also not implemented in a typed functional programming language, as is typical of most programmable proof-assistants, but is rather implemented in the decidedly *unsafe* systems programming language, Rust [17]. Note that this decision introduces no risk to system soundness, as Supervisionary's soundness ultimately rests on the continued separation of kernel-private data from Supervisionary's analogue of user-space – using privilege

[6] Barring unikernels, or library operating systems, like Mirage [20, 21]. If we are really pushing this analogy note that unikernels are in some respects quite similar to LCF-style proof-assistants in this regard, having their kernel linked with untrusted "user-space" and separated using programming language features like modules, rather than privilege and memory isolation.

[7] Many of the ideas presented henceforth are logic-independent. Though we have chosen to use HOL the ideas presented herein can be applied to a wide variety of other logics and type theories with relatively straightforward changes.

and private memories – and not on the type system of the implementation programming language. Moreover, as user-space and kernel communicate across a defined system call interface, untrusted user-space may also be written in *any* programming language capable of producing code that is binary-compatible with the Supervisionary kernel. Supervisionary therefore has no "metalanguage" in the LCF sense, but rather an implementation language, with automation potentially written in multiple languages – maybe even a mix.

For ease of implementation and use Supervisionary is implemented as a WebAssembly [12] (Wasm henceforth) host. We extend a Wasm virtual machine with new system calls that perform a context switch into Supervisionary, which has its own memory isolated from the memory of the executing user-space Wasm process running under its supervision, and inaccessible to it. This separation is only one way: the kernel can "peer in" to the runtime state of a running Wasm process and read from, or write to, its private memories. This decision means we may experiment with the fundamental ideas behind Supervisionary – namely isolating the kernel using private memory areas, the split between kernel- and user-space, a kernel system call interface – without becoming bogged down in extraneous detail associated with the booting ceremony of a real machine, for example. Moreover, we harness work on porting compiler and linker toolchains, allowing our user-space to be written in any programming language with a toolchain capable of targeting Wasm. Supervisionary's design will be fully described in Section 3.

Lastly, and more speculatively, Supervisionary's handles can be passed around a program, between different programs executing concurrently or sequentially under Supervisionary's management, or between the user-space program and the kernel. Whilst this property is not unique to Supervisionary – values of the abstract type of theorems may also be passed around within any LCF-style system, for example – the objects which these handles denote need not be necessary truths of pure mathematics, but can be contingent truths, themselves *functions* of the runtime state of the program itself, or of the Supervisionary kernel. Handles to these theorems act as a form of *capability*, in the computer security sense of that word. This property is unique to Supervisionary, as it rests on Supervisionary's dual status as a proof-assistant kernel, capable of generating and checking theorems, and an extension of a general purpose virtual machine, capable of executing arbitrary programs. Here, Supervisionary exploits its status as a "pinch point" that user-space cannot help pass through in order to have any sort of computational effect, to force user-space to pass a handle denoting a theorem that *proves* that it is acting correctly, per some system-wide policy. Some ideas of how this idea could develop are discussed later, in Section 4.

2 Implemented logic

Supervisionary implements a variant of Gordon's HOL [10], a classical higher-order logic. This can be intuitively understood as Church's Simple Theory of Types [7] extended with ML-style top-level polymorphism. We introduce the basics of this logic here, introducing just enough material that the unfamiliar reader can follow the rest of the paper.

We fix a denumerable set of *type variables* and use α, β, γ, and so on, to range arbitrarily over them. We work with *simple types* generated by the following recursive grammar:

$$\tau, \tau', \tau'' ::= \alpha \mid \mathsf{f}(\tau, \ldots, \tau')$$

Here f is a *type-former* which has an associated *arity* – a natural number indicating the number of type arguments that it expects. If all type-formers within a type are applied to a number of types matching their arity we call the type *well-formed* – that is, arities introduce a trivial or degenerate form of *kinding* for types. We will only ever work with well-formed

$$\dfrac{r : \tau}{\Gamma \vdash r = r} \qquad \dfrac{\Gamma \vdash r = s}{\Gamma \vdash s = r} \qquad \dfrac{\Gamma \vdash r = s \quad \Gamma' \vdash s = t}{\Gamma \cup \Gamma' \vdash r = t} \qquad \dfrac{\phi \in \Gamma}{\Gamma \vdash \phi} \qquad \dfrac{\Gamma \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi}$$

$$\dfrac{\Gamma \vdash r = s \quad \Gamma' \vdash t = u}{\Gamma \cup \Gamma' \vdash r\ t = s\ u} \qquad \dfrac{\Gamma \vdash r = s \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \lambda x_\tau.r = \lambda x_\tau.s} \qquad \dfrac{}{\Gamma \vdash \top}$$

$$\dfrac{\Gamma \vdash \phi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi \wedge \psi} \qquad \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \phi} \qquad \dfrac{\Gamma \vdash \phi \wedge \psi}{\Gamma \vdash \psi} \qquad \dfrac{\Gamma \cup \{\phi\} \vdash \psi \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi \longrightarrow \psi}$$

$$\dfrac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi} \qquad \dfrac{\Gamma \vdash \phi \quad \psi : \mathsf{bool}}{\Gamma \vdash \phi \vee \psi} \qquad \dfrac{\Gamma \vdash \psi \quad \phi : \mathsf{bool}}{\Gamma \vdash \phi \vee \psi} \qquad \dfrac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \psi}$$

$$\dfrac{\Gamma \vdash \phi \vee \psi \quad \Gamma' \cup \{\phi\} \vdash \xi \quad \Gamma'' \cup \{\psi\} \vdash \xi}{\Gamma \cup \Gamma' \cup \Gamma'' \vdash \xi} \qquad \dfrac{\Gamma \vdash \phi \longrightarrow \psi \quad \Gamma' \vdash \psi \longrightarrow \phi}{\Gamma \cup \Gamma' \vdash \phi = \psi}$$

$$\dfrac{\Gamma \vdash \exists x_\tau.\phi \quad \Gamma \cup \{\phi[x_\tau := y_\tau]\} \vdash \psi \quad y_\tau \notin fv(\psi) \cup fv(\Gamma) \cup \{x_\tau\}}{\Gamma \vdash \psi} \qquad \dfrac{\Gamma \vdash \phi = \psi \quad \Gamma' \vdash \psi}{\Gamma \cup \Gamma' \vdash \phi}$$

$$\dfrac{\Gamma \cup \{\phi\} \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \neg\phi} \qquad \dfrac{\Gamma \vdash \neg\phi \quad \Gamma' \vdash \phi}{\Gamma \cup \Gamma' \vdash \bot} \qquad \dfrac{\Gamma \vdash \forall x_\tau.\phi \quad r : \tau}{\Gamma \vdash \phi[x_\tau := r]}$$

$$\dfrac{\Gamma \vdash \phi[x_\tau := r]}{\Gamma \vdash \exists x_\tau.\phi} \qquad \dfrac{\Gamma \vdash \phi \quad x_\tau \notin fv(\Gamma)}{\Gamma \vdash \forall x_\tau.\phi} \qquad \dfrac{s : \tau' \quad r : \tau}{\Gamma \vdash (\lambda x_\tau.s)r = s[x_\tau := r]} \qquad \dfrac{\Gamma \vdash \exists x_\tau.\phi}{\Gamma \vdash \phi(\epsilon x_\tau.\phi)}$$

$$\dfrac{f : \tau \Rightarrow \tau' \quad x_\tau \notin fv(f)}{\Gamma \vdash \lambda x_\tau.(f\ x) = f} \qquad \dfrac{\Gamma \vdash \phi \quad r : \tau}{\Gamma[x_\tau := r] \vdash \phi[x_\tau := r]} \qquad \dfrac{\Gamma \vdash \phi}{\Gamma[\alpha := \tau] \vdash \phi[\alpha := \tau]}$$

Figure 3 The Natural Deduction relation for Gordon's HOL.

types in Supervisionary. We write $tv(\tau)$ for the *set of type-variables* appearing within a type, and write $\tau[\alpha := \tau']$ for the *type substitution* replacing all occurrences of α with τ' in the type τ. From the outset we assume two primitive type-formers built-in to the logic itself and necessary to bootstrap the rest of the material: bool, the type-former of the Boolean type and also the type of propositions, with arity 0, and $- \Rightarrow -$, the type-former of the HOL function space, with arity 2. Note we will abuse syntax and also write bool for the *type* of Booleans and propositions, and also write $\tau \Rightarrow \tau'$ for the function space type.

For each well-formed type τ we assume a countably infinite set of *variables* and *constant symbols*. We use x_τ, y_τ, z_τ, and so on, to range over the variables associated with type τ, and similarly use C_τ, D_τ, E_τ, and so on, to also range over the constants associated with type τ. With these, we recursively define *terms* of the explicitly-typed λ-calculus, as follows:

$$r, s, t ::= x_\tau \mid \mathsf{C}_\tau \mid rs \mid \lambda x{:}\tau.r$$

Note that there is an "obvious" simple-typing relation on terms, which is presented in Figure 4. We write $r : \tau$ to assert that a derivation tree rooted at $r : \tau$ and constructed according to the rules in Figure 4 exists, or more intuitively, that r has type τ. We call any term with a type *well-typed*; we will only ever work with well-typed terms in Supervisionary. Also, we call a term with type bool a *formula* and use ϕ, ψ, ξ, and so on, to suggestively range over terms that should be understood as being formulae in the rest of the paper. We work with

$$\frac{}{x_\tau : \tau} \qquad \frac{}{\mathsf{C}_\tau : \tau} \qquad \frac{r : \tau \Rightarrow \tau' \quad s : \tau}{rs : \tau'} \qquad \frac{r : \tau'}{\lambda x_\tau . r : \tau \Rightarrow \tau'}$$

Figure 4 The typing relation on terms.

terms identified up-to α-equivalence, write $fv(r)$ for the set of *free variables* of the term r, write $r[x_\tau := t]$ for the usual *capture-avoiding substitution* on terms, and write $r[\alpha := \tau]$ for the recursive extension of the type substitution action to terms.

As with type-formers, from the outset we assume a collection of typed constants needed to boostrap the rest of the logic, summarised in the table below:

$=$		$\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$
$\top$, $\bot$		bool
$\neg$		$\mathsf{bool} \Rightarrow \mathsf{bool}$
$\wedge$, $\vee$, $\longrightarrow$	*with type*	$\mathsf{bool} \Rightarrow \mathsf{bool} \Rightarrow \mathsf{bool}$
$\forall$, $\exists$		$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \mathsf{bool}$
ϵ		$(\alpha \Rightarrow \mathsf{bool}) \Rightarrow \alpha$

Most of the constants above are the familiar logical constants and connectives of first-order logic, lifted into our higher-order setting, and are introduced without further explanation. Only the ϵ constant – Hilbert's *description operator* [23], a form of choice – may be unfamiliar. In HOL, this can be used to "select", or "choose" an element of a type according to some predicate, and is otherwise undefined if no such element exists. Note therefore that all HOL types are inhabited by at least one element, with the term $\epsilon x_\tau.\bot$ inhabiting every type. We adopt conventional associativity, fixity, and precedence levels when rendering terms using these constants, writing $\phi \longrightarrow \psi$ instead of $(\longrightarrow\ \phi)\psi$, for example, and also suppress explicit type substitutions required to make terms involving polymorphic types well-typed, for example writing $\forall x_\tau.\phi$ instead of $\forall[\alpha := \tau](\lambda x_\tau.\phi)$.

We call a finite set of formulae a *context*, ranged arbitrarily over by Γ, Γ', Γ'', and so on. We write $\Gamma[x_\tau := r]$ and $\Gamma[\alpha := \tau]$ for the pointwise-lifting of the capture-avoiding substitution and type substitution on terms to contexts, and write $fv(\Gamma)$ for the set $\bigcup\{fv(r) \mid r \in \Gamma\}$. We introduce a dyadic *Natural Deduction relation* betwixt contexts and formulae using the rules in Figure 3, and write $\Gamma \vdash \phi$ to assert that a derivation tree rooted at $\Gamma \vdash \phi$ and constructed according to the rules presented in this figure exists.

Note that our Natural Deduction relation can be simplified following the equational treatment of the quantifiers and connectives discovered by Quine and Henkin, and implemented in the HOL Light proof assistant [14], a point we touch on later in Section 5. We prefer a more explicit treatment here, closer to a standard textbook presentation of Natural Deduction.

3 The Supervisionary kernel state

Supervisionary's kernel manages a series of *heaps*, or private memories, in addition to other bits of book-keeping data. These heaps contain *kernel objects*, of various kinds: type-formers, types, constants, terms, and theorems. These follow the progression of the different kinds of HOL objects and their interdependencies, as introduced in Section 2.

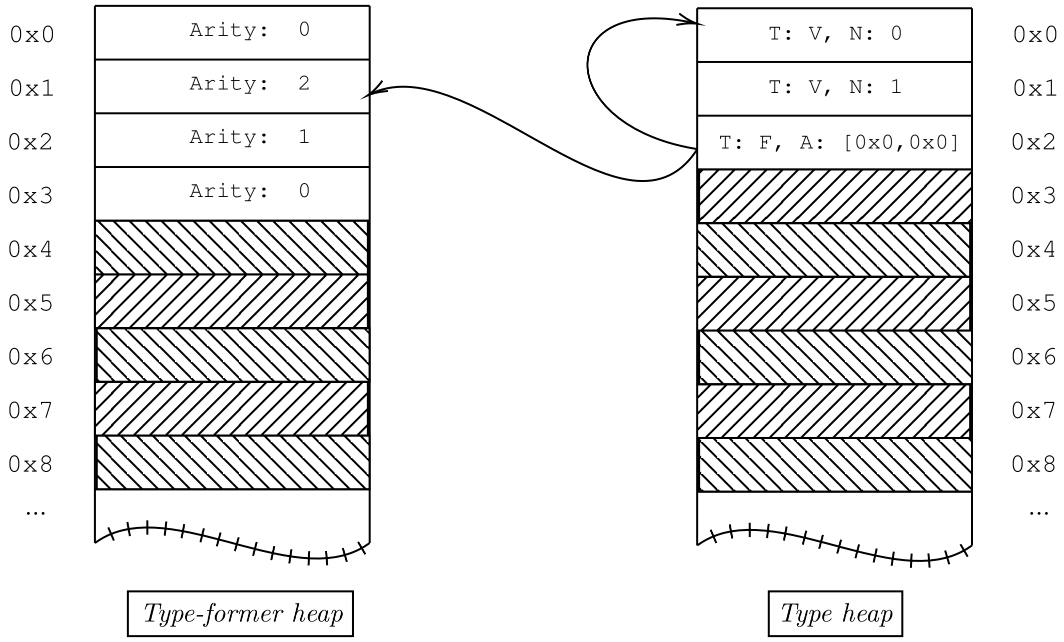


Figure 5 Entries within the Supervisionary kernel's type heap referencing entries within the type-former heap. Cross-hatched heap cells are as-yet unallocated by the kernel. The cell allocated at address `0x2` in the type heap is tagged with the `F` tag, indicating it is a type-former applied to a list of argument types, and points-to the cell at address `0x1` in the type-former heap, with arity 2. Two copies of the type stored in the cell with address `0x0`, containing a type-variable with name 0, are used as the argument of the type-former to produce a complete, well-formed type. Adopting the convention that type-variable α is at `0x0` in the type heap, and the function-space type-former $- \Rightarrow -$ is at `0x1` in the type-former heap, then this represents an encoding of the type $\alpha \Rightarrow \alpha$.

3.1 The type-former heap

The most foundational of all of the heaps is the heap of type-formers, which is manipulated and queried using a series of dedicated system calls. Each cell within the heap is either *unallocated* or *allocated* and, in the latter case, contains a natural number *arity* for a type-former, encoded as an unsigned 64-bit machine word. New type-formers are registered within the heap by invoking a dedicated system call from user-space – `TypeFormer.Register` – which takes as input the arity of the type-former and in response allocates a fresh cell, returning the address of the cell back to user-space as the output of the system call. This address is the handle to the new type-former kernel object, now under management by the Supervisionary kernel, and must be used by user-space to refer to this object henceforth. For example, a handle can be passed to the system call `TypeFormer.IsRegistered` system call to test whether a handle denotes a registered type-former. Alternatively, the `TypeFormer.Resolve` system call can be used to *dereference* a handle, in order to obtain an arity, providing that it does indeed denote a registered type-former, otherwise returning a defined error code.

Note that type-formers are essentially "named" by their handle: there may be many type-formers with the same arity registered with the kernel, and the particular meaning of any type-former is largely a convention of user-space, outside the purview of Supervisionary. Two primitive type-formers, pre-registered in the type-former heap on system boot, are however exceptions to this rule and hold special significance for the kernel. These are the `bool` type-former, registered at address `0x0` with arity 0, and the function-space type-former

$\Rightarrow$, registered at address `0x1`, with arity 2. The existence of these type-formers must be understood by user-space, as they form part of the Supervisionary system interface, similar to how the distinguished file handles `stdout` and `stdin` are part of the POSIX system interface and must be understood by user-space and kernel alike to support file I/O.

3.2 The type heap

Building atop the heap of type-formers is the heap of types, queried and manipulated using another series of system calls, with the interface for working with types much more complex than that for type-formers. As a result, it is only summarised here.

Recalling Section 2, types are either a type-variable or a *combination* of a type-former applied to a list of types. All entries within the type heap are therefore tagged indicating whether they are a type-variable or a combination. Type-variable entries contain one datum: the *name* of the type-variable, an unsigned 64-bit machine word. Combination entries also contain a pointer into the type-former heap, indicating which type-former is being applied, and contain a list of pointers back into the type heap itself, identifying the type arguments of the combination. Figure 5 shows a schematic diagram of dependencies between cells within the two heaps, wherein we use `V` to tag type-variables and `F` to tag combinations.

Supervisionary also boots with some entries in the type heap pre-registered, corresponding to common or useful types used to boostrap the rest of the logic. These include the Boolean type, bool, common type variables – α and β, for example – as well as larger, more complex types such as the type of the polymorphic equality, $\alpha \Rightarrow \alpha \Rightarrow \mathsf{bool}$. The handles for all of these pre-registered entities must likewise be understood by user-space.

Further derived types, built from primitive objects or otherwise, may be built using `Type.Register.Variable` and `Type.Register.Combination` system calls for constructing basic types. The first takes as input only a 64-bit machine word – the name of the variable – and immediately registers a new type in the type heap, returning the newly-allocated handle. On the other hand, `Type.Register.Combination` takes as input a handle pointing-to a registered type-former in the type-former heap and a list of handles pointing back into the type heap. The system call fails if any of these handles dangle, or denote an object of the wrong kind, or if a list of type handles is presented with a length differing from the registered arity of the type-former. Lists of handles are passed to system calls by passing a base pointer, denoting the beginning of the list (or rather, array) with an explicit length. Substitutions, for the `Type.Substitution` system call, which performs a type-substitution, are passed as two lists: one for the domain of the substitution, another for the range.

It is sometimes convenient to test the structure of a type pointed-to by a handle. This can be done using system calls like `Type.Test.Combination` which takes a handle and returns a Boolean value indicating whether the corresponding type is a combination. A family of “splitting” system calls – `Type.Split.Variable`, for example – can also be used to deconstruct a type. This takes a handle and returns the name of the variable pointed-to by the handle, if it is indeed a type-variable. Similar functions also exist for type combinations, and allow user-space to “pattern match” on types.

A system call, `Type.Variables`, also exists for computing the type-variables appearing within a term. Implementing this as a system call is a challenge as the number of variables to be returned – and hence the size of buffer that user-space needs to set aside to hold them, and which Supervisionary will write into – is unpredictable. To resolve this, the kernel exposes another system call, `Type.Size`, which computes the *size* of a type which bounds the number of variables appearing within a type. By querying this, user-space can first allocate sufficient memory within its own address space to hold the set of type-variables before calling `Type.Variables` with a pointer to the base of the allocated buffer.

Obviously, the Supervisionary kernel must be careful in its management of its heaps, and this topic becomes pressing now we have introduced two heaps with dependencies between them. In particular, Supervisionary maintains a series of *kernel invariants* which hold immediately out of boot and must be preserved by all system calls. One key invariant is the idea that heaps only ever *grow* monotonically, and allocated entries are immutable. Once an object is allocated into the heap it cannot be removed or modified in any way, lest we introduce an unsoundness, for example by modifying the bool type, or the truth constant, $\top$, or something similarly catastrophic. Moreover, heaps should always remain *inductive*, in the sense that their cells do not contain any dangling pointers that do not point-to allocated cells in the same or other heaps. Essentially, this latter property forces the various objects under Supervisionary's management to correctly follow the grammar of types and terms introduced in Section 2, with larger objects being gradually "built up" out of smaller ones.

3.3 The constant and term heap

Building on the heap of types is the heap of constants, keeping track of registered term constants. Again, this is pre-provisioned with a series of primitive constants, corresponding to the logical constants and connectives, at boot-time. The system call interface for constants is similar to that for type-formers, exposing just three system calls for registering new constants, dereferencing handles, and testing whether a handle denotes a registered constant.

Another, further heap – the heap of terms – is also used to construct and manipulate terms, with heap cells tagged with whether they represent a variable, constant, application, or lambda-abstraction, in a similar style to the tagging used for cells in the type heap. System calls for constructing, testing, and pattern matching on terms are provided, similar to those previously discussed within the context of other heaps. Further, new special-purposes system calls, for example `Term.Type.Infer` allow user-space to infer the type of a registered term, if any, whilst `Term.Substitute` performs a capture-avoiding substitution on a term. Note that handles for terms actually denote α-equivalence classes of terms – at present, we use a name-carrying syntax, but could implement this using De Bruijn indices or levels [8], leading to a more efficient implementation.

3.4 The theorem heap

The final, and most important heap maintained by the Supervisionary kernel is the heap of theorems. Every other Supervisionary heap exists to support this heap, and Supervisionary considers a theorem proved only if it appears in this heap. Cells within the theorem heap contain a *sequent*, a tuple consisting of an (ordered) set of handles of formulae, representing the assumptions of the theorem, combined with a single handle for the theorem's conclusion.

A theorem kernel object can be deconstructed using the `Theorem.Split.Assumptions` and `Theorem.Split.Conclusion` system calls, to obtain the list of assumption and conclusion of the theorem object, respectively. However, the only way that a new entry in the heap of theorems can be constructed is by using one of a series of system calls corresponding to an inference rule of the logic's Natural Deduction relation, presented in Section 2, or of the definitional principles of HOL. Taking the *negation introduction* system call, for example:

$$\frac{\Gamma \cup \{\phi\} \vdash \bot \quad \phi : \mathsf{bool}}{\Gamma \vdash \neg\phi}$$

We have a corresponding system call `Theorem.Register.Negation.Introduction` which takes a handle pointing-to a sequent, $\Gamma \cup \{\phi\} \vdash \bot$, in the kernel's theorem heap, and a handle pointing-to a term, ϕ, in the kernel's term heap, and returns a handle pointing-to a theorem, $\Gamma \vdash \neg\phi$, also residing in the kernel's theorem heap if all error checks pass for the inputs.

Like terms, theorem handles point-to α-equivalence classes of theorem objects, wherein two sequents are considered the same if their respective constituent handles point-to the same α-equivalence classes of terms. Moreover, the Supervisionary kernel also enforces *maximal sharing* in all of its kernel heaps, and an attempt to register an object that has already been registered, up-to α-equivalence, does not allocate a new slot in the respective kernel heap, but merely returns the existing handle to the object. These two decisions make some operations within the Supervisionary kernel easier to implement, at the expense of slowing down the registering of new objects. For example, we know that objects are α-equivalent when their handles are identical. Moreover, in `Theorem.Register.Negation.Introduction` above, we know that the formula ϕ is not in the context $\Gamma \cup \{\phi\}$ if the second input handle, mentioned above, does not appear in the list of handles representing the assumptions of the sequent. Note that this would not be the case if we did not enforce maximal sharing: *another* handle pointing-to the term ϕ may be present in the list of assumptions of the theorem, different from the handle passed in from user-space, and this would force Supervisionary to have to perform a "deep scan" of its heaps in trying to work out whether the two handles supposedly pointed-to the same HOL formula. As a result of this sharing, Supervisionary's heaps remain inductive in the sense previously discussed, but recursively-defined objects represented within them are not necessarily encoded as trees, but rather directed acyclic graphs.

Moreover, we previously mentioned that heaps must continue to grow monotonically at all times, lest we inadvertently introduce an unsoundness into the system by allowing the HOL bool type, or similar, to be redefined. However, note that this invariant *could* be weakened, somewhat, by "working backwards" from the kernel's theorem heap and removing objects in other kernel heaps that are not referenced via a transitive points-to relation. Essentially this would represent a form of *mark-and-sweep garbage collection* [31] wherein objects in the kernel's theorem heaps are root objects, with other objects deallocated if they are not reachable from these roots. Care must be taken to ensure that the primitive kernel objects, pre-provisioned into the heaps at system boot, can never be deallocated, even if currently unreachable. Whilst possible, this garbage collection process is not at present implemented in Supervisionary, as sharing compresses the heaps with no pressing need to remove objects from them. Moreover, within the context of garbage collection, user-space cannot be sure that a handle generated by the kernel, and previously denoting a registered kernel object, is stable and now does not dangle, complicating the Supervisionary programming model.

3.5 Specifying kernel functions

Implementing and using the Supervisionary kernel is an extended exercise in heap and pointer manipulation, and until now the kernel's system calls were explained in an intuitive, informal sense. To specify the behaviour of some of our kernel system calls, we therefore reach for an existing tool used to specify pointer-manipulating programs: Separation Logic [30, 16].

Working abstractly, we represent handles as elements of the set $\mathbb{N}$ of natural numbers, and use h, h', h'', and so on, to range over handles. For a fixed set A, we say that a partial-function $f : \mathbb{N} \rightharpoonup A$ is *finitely-supported* when the set $dom(f) = \{x \mid f\ x \text{ defined}\}$ is finite. We call such a finitely-supported partial map into a set A an *A-heap*. We write 0 for an empty A-heap, and for two A-heaps f and g we write $f \sharp g$ to assert that their domains are disjoint, so $dom(f) \cap dom(g) = \{\}$. This relation is symmetric and $0 \sharp g$ always, for any g. Moreover, for two A-heaps f and g we can "glue them together", using the function $f \oplus g$, to form a larger A-heap. This function is defined piecewise as:

$$\begin{aligned}
&(f \oplus g)\ x = f\ x \text{ if } x \in \mathit{dom}(f)\\
&(f \oplus g)\ x = g\ x \text{ if } x \in \mathit{dom}(g)\\
&(f \oplus g)\ x \text{ is undefined otherwise}
\end{aligned}$$

Note that $f \oplus g$ is well-defined whenever $f \sharp g$. Finally, for $a \in A$, we write $h \mapsto a$ for the *singleton A-heap* mapping h to a and remaining undefined at all other points.

We define *types*, *constants*, *terms*, and *theorems* by the following non-recursive grammars, where m ranges over arbitrary natural numbers:

$$\begin{aligned}
t, t', t'' &::= \texttt{TyVar}\ m \mid \texttt{TyFm}\ h\ (h_1, \ldots, h_n)\\
C, C', C'' &::= \texttt{TConst}\ h\ h'\\
r, r', r'' &::= \texttt{Var}\ m\ h \mid \texttt{Const}\ h\ h' \mid \texttt{App}\ h\ h' \mid \texttt{Lam}\ m\ h\ h'\\
s, s', s'' &::= \texttt{Seq}\ (h_1, \ldots, h_n)\ h
\end{aligned}$$

We call heaps over types a *type-heap*; similarly for constants, terms, and theorems. We also call heaps over natural number arities a *type-former heap*.

Fix a set of kernel states K. We use k, k', k'', and so on, to range over kernel states, each of which is a 5-tuple $\langle F, Ty, C, Tm, Th\rangle$ consisting of a type-former heap, a type heap, a constant heap, a term heap, and a theorem heap respectively. We extend our notion of disjointness to kernel states, and write $k \sharp k'$ to assert that all of the respective heaps in kernel states k and k' are disjoint. We further abuse notation and write 0 for the *empty kernel state* consisting of five empty heaps, and $k \oplus k'$ for the "gluing" of two kernel states together, wherein each of the respective heaps in k and k' are joined pointwise using $\oplus$. Note that, again, $k \oplus k'$ is well-defined whenever $k \sharp k'$.

We define *assertions* as sets of kernel states, use A, B, C, and so on, to range over them, and write $k \vDash A$ to assert that $k \in A$. We pay especial attention to some particular assertions of note that will be useful in specifying some of our system calls:

$$\begin{aligned}
\bullet &\equiv \{\langle 0, 0, 0, 0, 0\rangle\}\\
A \star B &\equiv \{k'' \mid \exists k\ k'.k'' = k \oplus k' \text{ and } k \sharp k' \text{ and } k \vDash A \text{ and } k' \vDash B\}\\
h \mapsto_{\text{Aty}} a &\equiv \{\langle h \mapsto a, 0, 0, 0, 0\rangle\}\\
h \mapsto_{\text{Typ}} t &\equiv \{\langle 0, h \mapsto t, 0, 0, 0\rangle\}
\end{aligned}$$

We further define the standard logical constants and connectives as abbreviations for setwise operations, writing $\bot$ for $\{\}$, $C \wedge D$ for $C \cap D$, and $\exists x.C\ x$ for $\bigcap_x . \ C\ x$, for example.

Fix a set of *values*, V, consisting *at least* of handles and numeric error codes. System calls e, f, g, and so on, are modelled as total functions from kernel states to kernel states which also produce a value as a side-effect, that is $e : K \to V \times K$. Note that though a kernel system call may fail – for example, if its inputs are in an unexpected form, or similar – it should never *crash*, but rather return a specific error code back to the user-space program and maintain the state of the kernel as it was before the system call was invoked. Crashes, or *kernel panics*, are reserved for unrecoverable errors, for example the failure of an internal invariant, or similar – the Supervisionary equivalent of a "blue screen of death".

With this in mind, we define a Separation Logic triple as a three-place relation between an assertion, a system-call, and a function from values to assertions by:

$$A \vdash e \dashv \lambda r.B \text{ iff for any } C \text{ if } k \vDash A \star C \text{ and } e\ k = \langle v, k'\rangle \text{ then } k' \vDash (\lambda r.B)v \star C$$

With this, we specify the behaviour of the `TypeFormer.Register` system call as follows:

- $\vdash \texttt{TypeFormer.Register}(a) \dashv \lambda h.h \mapsto_{Aty} a$

Note that this specification correctly captures the fact that the call can never fail: it will always return a handle pointing-to a new cell in the type-former heap, containing the required arity, with no other effects on the kernel heaps.

Specifying system calls which manipulate types, constants, terms, or theorems is more complex as we must assume that any handles contained within these structures point-to allocated cells in an appropriate kernel heap. To do this, we use of a family of *shape predicates* relating encodings of objects within the kernel's heaps to the recursively-defined structures of Section 2. Assuming a bijection V between natural numbers and type-variables, and a bijection F between handles and type-formers, we inductively define the relation $\texttt{TYPE}\ h\ \tau$:

$$\frac{h \mapsto_{Typ} \texttt{TVar}\ m \quad (V\ m\ \alpha)}{\texttt{TYPE}\ h\ \alpha}$$

$$\frac{h \mapsto_{Typ} \texttt{TyFm}\ h'\ (h_1, \ldots, h_n) \star h' \mapsto_{Aty} n \star \texttt{TYPE}\ h_i\ \tau_i \quad (1 \leq i \leq n, F\ h'\ \mathsf{f})}{\texttt{TYPE}\ h\ \mathsf{f}(\tau_1, \ldots, \tau_n)}$$

We omit comparable shape predicates for constants, terms, and theorems, as the pattern should be clear. Note that the basic allocation functions for types, upon success, generate kernel states wherein the `TYPE` relation holds. For example, assuming a correspondence, $V\ n\ \alpha$, between the natural number n and type-variable α:

- $\vdash \texttt{Type.Register.Variable}(n) \dashv \lambda h.\texttt{TYPE}\ h\ \alpha$

Similarly, we have:

$$\begin{aligned} &h \mapsto_{Aty} n \star \texttt{TYPE}\ h_1\ \tau_1 \star \ldots \star \texttt{TYPE}\ h_n\ \tau_n \\ \vdash\ &\texttt{Type.Register.TypeFormer}(h, h_1, \ldots, h_n) \dashv \\ \lambda r.&h \mapsto_{Aty} n \star \texttt{TYPE}\ h_1\ \tau_1 \star \ldots \star \texttt{TYPE}\ h_n\ \tau_n \star \texttt{TYPE}\ r\ \mathsf{f}(\tau_1, \ldots, \tau_n)\end{aligned}$$

Which also captures the fact that existing well-formed kernel heaps remain well-formed after invocation of a system call, with shape predicate invariants formally capturing the kernel invariants previously informally introduced.

3.6 Programming in user-space

The system call interface presents a very low-level, austere interface to user-space code. To make programming Supervisionary less tedious, a utility library, similar in function to `libc`, is provided to user-space in order to raise the level of abstraction above the raw system call interface. This is provided as `libsupervisionary`, currently implemented only for the Rust programming language, but could in theory be ported to the C-language, or any other language that can be compiled to Wasm. Note that further layers, built on top of `libsupervisionary`, can provide pretty-printing and parsing routines for types and terms, automation, proof-state management, and other functions typical of a proof-assistant.

4 Future work

We now take a more speculative turn, discussing future work. The ideas presented in this section are perhaps the most interesting consequence of Supervisionary's design, and we therefore dedicate a section solely to them.

4.1 Capabilities on steroids

As described, Supervisionary is a proof-checking system implemented in an unusual way, but also a virtual machine, capable of executing arbitrarily complex programs compiled to the Wasm instruction set, from a variety of source programming languages.

However, at present, these Wasm programs are limited in the *effects* that they can make on the system – specifically, the only effect that they can actually make, other than heating the CPU, is to construct types, terms, and theorems, in Supervisionary's various heaps, using the series of system calls progressively introduced in Section 3. Programs executing under Supervisionary are so-far incapable of opening files on the user's machine, communicating over sockets, or querying the system time, because Supervisionary does not provide any system calls to allow a program to perform any of those activities. However, it could.

Specifically, Supervisionary could implement a system interface that provided all of the system calls needed by "real" programs wishing to make some effect on a user's machine. By doing this, Supervisionary is transformed into a general-purpose virtual machine, akin to the Java Virtual Machine, capable of executing arbitrary programs – calculators, simulations, file search utilities, and so on – albeit with a bizarre set of extra system calls dedicated to theorem proving. In short, by extending Supervisionary with system calls for querying and manipulating the system state, Supervisionary is *both* a proof-assistant and a general-purpose virtual machine – though these two facets of the system are kept separate.

These two families of system call need not be kept separated, however. Prior to allowing a user-space program to open or read a file, Supervisionary could first demand that a (handle to a) theorem is supplied to it as an extra argument to the file-open system call, `fopen`, for example. Interestingly, because Supervisionary executes at a relative level of privilege, and can "peer in" to the runtime state of a user-space program, the statement of this desired theorem can be a *function* of the runtime state of the user-space program itself, of the runtime state of the Supervisionary kernel, and also of the various arguments and other details of the system call being invoked. This statement – which we will call the *challenge* – can be any arbitrary formula written in HOL, and can be generated dynamically by the kernel, perhaps in accordance with a *global policy* enforced by Supervisionary. A failure to produce a handle to address a particular challenge causes the system call to fail, with a runtime failure.

For concreteness, suppose we fix HOL types wstate, kstate, and cstate, which you may imagine as being record types capturing details of the runtime state of the executing Wasm process, the runtime state of the Supervisionary kernel, and the details of the system call being invoked. Supervisionary can dynamically *reflect* the actual runtime states of the user-space program and kernel, and the invoked system call, into inhabitants of these HOL types. Then, supposing our prevailing security policy, p, is a HOL function of type $\mathsf{wstate} \Rightarrow \mathsf{kstate} \Rightarrow \mathsf{cstate} \Rightarrow \mathsf{bool}$, a challenge is obtained by dynamically applying p to the reflected records, described above. Two particularly special security policies exist:

$$\lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\bot \qquad \text{and} \qquad \lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\top$$

When applied to a reflected runtime state, these two policies generate the challenges $\bot$ and $\top$, respectively. The first policy is therefore the *deny all* policy, which essentially prevents a user-space program from invoking *any* system call, and making any effect on the system state, whilst the second is the *allow all* policy which can always be trivially satisfied by passing the handle to HOL's truth introduction theorem.[8] Between these two extremal points are

[8] Note that, if we allow arbitrary axioms to be introduced into the Supervisionary global theory, as many

a variety of other interesting policies, however. For example, if we assume that our cstate record contains a field cname of type cstate ⇒ string capturing the name of the system call being invoked, then we may selectively prevent particular system calls from being executed by a program. The following policy prevents any invocation of the `fopen` and `fclose` system calls from succeeding, for example:

$$\lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\mathsf{cname}\ c \notin \{\mathtt{fopen}, \mathtt{fclose}\}$$

This policy is expressible using existing security mechanisms on mainstream operating systems: modern Linux distributions use small eBPF programs to block programs from invoking particular system calls at runtime, according to a security policy, for example. However, the mechanism sketched above goes far beyond the expressivity of these existing systems as *correctness* properties can also be captured by a policy, for example. Assume, for example, that the cstate record also exposes a field cargs of type cstate ⇒ nat ⇒ option list word 8, which returns the byte-representation of the n^{th} argument passed to the invoked system call. With this, and assuming HOL functions strbytes and intbytes for converting string and machine word datatypes into byte lists, respectively, we can then express

$$\begin{aligned}&\lambda w_{\mathsf{wstate}}.\lambda k_{\mathsf{kstate}}.\lambda c_{\mathsf{cstate}}.\mathsf{cname}\ c = \mathtt{fwrite} \longrightarrow\\ &\quad \mathsf{cargs}\ c\ 0 = \mathsf{Some}\ (\mathsf{strbytes}\ \text{"foo.txt"}) \longrightarrow\\ &\qquad \mathsf{cargs}\ c\ 1 = \mathsf{Some}\ (\mathsf{intbytes}\ (\epsilon i_{\mathsf{word}\ 64}.3i^2 - 2i - 1 = 0))\end{aligned}$$

preventing any write to a file unless the 64-bit machine word being written is *some* zero of a particular polynomial. In particular, the policy above demonstrates an important point: Supervisionary's policies can use any aspect of HOL, quantifiers, choice, and all.

Until now, all examples have focussed on the cstate record which captures information about the invoked system call. Other interesting policies can also be written in terms of the runtime state of the Supervisionary kernel itself. This idea becomes especially interesting if we extend the kernel with new structures recording aspects of a program's execution. By extending Supervisionary to keep a log of all system calls invoked thus far by a user-space program – for example, exposing this log as a field wlog in the wstate record with type wstate ⇒ nat ⇒ option event – we can capture *trace properties* of the executing program. For instance, one may assert that system call invocations must be balanced in some way – exactly one file may be opened at a time, and opening a second file first requires the program close the other, for example – and also deeper properties, including adherence to a protocol.

One common security pattern deployed by software is gradual *jailing*, or shedding of capabilities – for example, OpenBSD's `pledge` system call allows a program to dynamically shed the ability to further invoke particular classes of system call, gradually dropping capabilities during a self-jailing phase. To offer a similar facility for Supervisionary, we need to allow a program to dynamically *strengthen* the prevailing policy being enforced by Supervisionary. Given the prevailing policy p we can allow the user-space program to self-jail by switching to a new policy q if the program can *prove* to Supervisionary that the new policy is more restrictive than the previous one, in the sense that:

$$\forall w_{\mathsf{wstate}}.\forall k_{\mathsf{kstate}}.\forall c_{\mathsf{cstate}}.q\ w\ k\ c \longrightarrow p\ w\ k\ c$$

proof-assistants allow, then we need some form of *taint tracking* to ensure that challenges may only be answered by theorems deduced without axioms.

If we view Supervisionary's policies as identifying sets of possible system behaviours, then the user-space program must prove to Supervisionary's satisfaction that the set of permissible behaviours that may occur from now on are a subset of the behaviours that Supervisionary was previously happy to accept. Note here that *quantification* is used in an essential way.

The material in this section has some similarity with an existing idea: *proof-carrying code* [24]. In one model of proof-carrying code the operating system or virtual machine loader is modified to check proof certificates bundled with binaries for adherence to some security or correctness property, for example memory safety, before the binary is executed. Note, however, that these certificates are constructed *up front*, in a separate step, and merely checked by the operating system loader. In contrast, the ideas presented above are more akin to *proof-generating code*, wherein the user-space program and Supervisionary work together to dynamically come to an understanding that the runtime behaviour of the program adheres to a prevailing policy. In effect, HOL is used as a *lingua franca* used to communicate demands by, and intent of, the Supervisionary kernel and user-space program, respectively.

The ideas above also blur the lines between static and dynamic, or runtime, verification. Supervisionary can be used like any other proof assistant, to statically establish properties of models of software or hardware systems, or reason about necessary truths within the rarefied domain of pure mathematics. However, it may also be used to dynamically check the runtime behaviour of programs executing under its supervision, interestingly also using theorem proving. Moreover, Supervisionary allows *any* program written in any programming language to be endowed with support for theorem-proving, and reasoning about its own behaviour. Indeed, a program executing on the Supervisionary virtual machine *must* be prepared to explain its adherence to the system security or correctness policy in order to have any hope of performing a side-effect. With Supervisionary, proof is no longer the exclusive domain of dedicated programming languages like Agda [26] or Idris [6, 5], but can be extended to any language merely by porting `libsupervisionary`.

4.2 Hardware-accelerated proof-checking

As noted earlier, from the perspective of user-space software a system call presents as a suite of particularly CISC-like machine instructions with a rather unorthodox method of invocation. Indeed, the combination of the Supervisionary system calls and the host Wasm instruction set can be, itself, thought of as a new, derived instruction set extending Wasm, with strange new domain-specific instructions for proof construction and management. Moreover, it should be quite clear that there is nothing Wasm-specific about Supervisionary, and indeed Wasm was chosen merely as a relatively pain-free way of experimenting with the core ideas behind Supervisionary. Indeed, Supervisionary could have been implemented as real, privileged systems software for an existing instruction set in a relatively straightforward manner.

As a result, the Supervisionary system call interface is already quite well-suited to an implementation in hardware, perhaps as an extension of an existing instruction set architecture like Arm AArch64 or RISC-V. The mechanism through which the Supervisionary kernel isolates itself, via private memories, is rather "hardware like", and maps nicely onto existing hardware features, and whilst the present Supervisionary system call interface makes extensive use of "pointer-like" handles to refer to kernel objects, on a real hardware implementation these handles could *literally* be pointers into private memories, or similar. Moreover, the system call interface itself is also further carefully designed to avoid arbitrarily large recursive structures, difficult for an instruction set architecture to handle, from being passed across the kernel system call boundary. We could therefore "push Supervisionary down one layer" again, into the underlying instruction set implemented by hardware. With this, the ideas presented in Subsection 4.1 take on a new light, as the system hardware is now capable of expressing, and enforcing, arbitrarily complex security and correctness properties.

5 Conclusions

5.1 Related work

The closest related work to Supervisionary is *VeriML*, an ML-like language extended with limited dependent-types ranging over HOL terms and theorems [34]. Essentially, VeriML "internalises" a typical HOL kernel implementation within a higher-order programming language, promoting the abstract type of theorems – typically *defined* within the system metalanguage – into a native type of the language that can be queried and modified with new, dedicated, domain-specific expressions for theorem construction and manipulation.

Compared to a typical HOL kernel, VeriML essentially "pushes the kernel down one layer" in the hierarchy of abstractions, moving the kernel from a library within the language to a first-class programming language feature. However, Supervisionary "pushes" the kernel even further, moving support for theorem proving out of the programming language and into the underlying operating system – or, in our case, virtual machine. (And, as discussed above, this "pushing" of the kernel down through the different layers of abstractions can be taken to its logical conclusion, by pushing the kernel all the way into hardware.) Note, however, that despite the general idea behind the two projects being essentially the same, the two differ markedly in a myriad of design details which have some important consequences: for example, automation in Supervisionary is inherently programming-language agnostic, whereas VeriML is inherently tied to one particular language – VeriML itself.

Interestingly, some of the ideas used in Supervisionary can also be "pushed up one layer" in the hierarchy of abstractions. Specifically, the Separation Logic specifications presented in Subsection 3.5 can be re-interpreted as a series of *local axioms* describing the behaviour of statements or expressions in a programming language for registering, manipulating and querying type-formers, types, and other objects, in a series of heaps secreted from the user, managed by the language's runtime. Interestingly, the natural programming language that one obtains from this exercise is imperative, in contrast to the functional VeriML. (To make a more ergonomic programming language it would make sense for these expressions to be modified so that they manipulate built-in recursive types of the programming language – corresponding to HOL type-formers, types, constants, terms, and theorems – in a similar fashion to VeriML, rather than make use of Supervisionary's handles and its incremental construction of recursive structures.)

In Subsection 4.1 we observed that Supervisionary's handles can be reinterpreted as *capabilities*, in the information security sense of that word. Note that capability machines are, at the present time, having a minor renaissance, driven by the success of the CHERI capability extensions for MIPS, Arm AArch64, and RISC-V [25]. Capabilities in hardware have a long and storied history – dating at least to the Cambridge CAP machine developed in the 1970s – and capability-based security has also previously been applied to programming languages and software, including systems software like operating systems. Whilst contemporary operating systems like seL4 [19] and other L4 derivatives have a security model built around capabilities, perhaps the best well-known historical example of a capability-based operating system was KeyKOS [29, 13] and its many derivatives, including EROS, the Extremely Reliable Operating System [32]. However, despite this long history, the Supervisionary conception of capabilities differs from other implementations as hardware-based capability systems like CHERI, are relatively inexpressive, merely extending traditional pointer types with information on valid memory regions within which they may point, and memory access permissions. This is because existing hardware-based capability systems are optimised to prevent spatial and temporal memory safety issues, inherent in the use of unsafe systems programming languages like the C-language, and derivatives, and must provide an easy “on ramp” allowing existing software to adopt them. Supervisionary’s conception of capabilities differs, here, in being more expressive, allowing complex security and correctness properties to be expressed, but also much more intrusive, and much harder to make use of: software must be aware of the prevailing security or correctness policy in force at the time, when trying to open a file for example, in order to be able to correctly answer the “challenge”. Using a Supervisionary capability to open a file may also require unbounded amounts of reasoning first, in order to address the “challenge” posed by Supervisionary, which is not the case with other forms of capability, which act as passive tokens of authority.

Lastly, Supervisionary, as an implementation of HOL, is closely related to several extant systems in the wider HOL family: Isabelle/HOL, HOL4, HOL Light, Candle [1], and so on. The kernels of all of these systems implement very similar logics, albeit with minor modification. However, unlike the aforementioned systems, Supervisionary does not follow the typical LCF-style of system organisation, nor is it written in an ML-derivative.

5.2 On trust

The current Supervisionary proof-checking kernel consists of approximately 6,600 lines of Rust code, compared to approximately 3,100 lines of Standard ML in recent distributions of the Isabelle framework. Whilst comparing linecounts between languages is an imperfect science, the Supervisionary kernel is still clearly larger than one of the most complex extant LCF-style implementations. Largely, this is because Supervisionary implements the HOL Natural Deduction relation in full, providing introduction and elimination rules for all logical constants, as observed in Section 2. Using a bare-bones implementation of HOL, based upon equality, and then deriving introduction and elimination rules for all other logical constants outside of the kernel, would be one way to shrink the kernel line count.

From the point-of-view of a Supervisionary user-space program the kernel is not the only body of code that must be trusted. The implementation of `libsupervisionary` – a mediation layer between kernel and user-space – must also be trusted in much the same way that `libc` must also be implicitly trusted by user-space on Unix-derivatives. A malicious `libsupervisionary` could present as interfacing with the kernel whilst maintaining shadow copies of kernel state, never requesting that the kernel actually check a proof, for example! This threat is not unique to Supervisionary – despite oft-repeated claims that the kernel represents the entire system TCB, users of all interactive theorem proving systems implicitly trust the system’s pretty-printer, for example, not to lie about what the system has proved [28], despite this code typically residing outside of the kernel – though the fact that there is a nuanced *threat model* here is perhaps more obvious, and pressing, as a consequence of Supervisionary’s dual status as proof-checker and general-purpose virtual machine. Minimising the system kernel size potentially comes at the cost of bloating other code – for example, `libsupervisionary` – that must also be trusted by users wishing to use the system for theorem proving tasks. However, the Supervisionary kernel may also contain functionality – filesystems, timers, network sockets, and similar – related to the Supervisionary general-purpose virtual machine and there is a danger that this code can be used by a malefactor to *exploit* the kernel, perhaps undermining the Supervisionary capability system by somehow deriving a contradiction in an empty context, or similar. On balance – and focussing on security and consistency, rather than efficiency – the kernel size should be minimised if possible, as this prevents security exploits and simply moves code that must be trusted for theorem-proving purposes around, neither helping nor hindering the system’s trust story in that respect.

Lastly, the kernel can also be modularised – and is, in the Supervisionary implementation – with all theorem-proving related material isolated inside its own module, all filesystem-related material within its own module, and the two only interacting when strictly needed. Ironically, this re-introduces the idea of protecting key system invariants using programming-language modules and type-abstraction albeit this is never directly exposed to the user.

5.3 Closing remarks

We have presented Supervisionary, a kernel for an implementation of Gordon's HOL. In contrast to most implementations of HOL, Supervisionary is not based on the LCF architectural pattern, but is instead implemented in a style more reminiscent of a typical operating system, making essential use of machine-oriented notions of separation to protect the system kernel from untrusted automation.

The Supervisionary kernel – which is open-source, and developed in the open[9] – is implemented as a host for the Wasmi interpreter[10] for Wasm. Interpretation means that software executing under Supervisionary executes orders of magnitude slower than natively-compiled code. However, the kernel is architected in a layered manner, with all important kernel functionality implemented in a library that is independent of the execution engine used and bound to the execution engine in a thin shim layer sitting between it and the core kernel library. As a result, Supervisionary can be ported to more efficient Wasm execution engines – the Wasmtime just-in-time compiler[11], for example – relatively easily. This porting has already started and will provide a significant increase in system performance, albeit at the cost of bringing a state-of-the-art just-in-time compiler into the kernel.

The design of Supervisionary is interesting in its own right: it completely dispenses with the typical metalanguage associated with an LCF-style proof-assistant, allowing automation to be written in any programming language capable of respecting the Supervisionary kernel binary interface and calling conventions. However, in our view the most interesting aspects of Supervisionary are the consequences of its design, and the possibilities for future work. These include adopting the Supervisionary kernel interface as the foundation of a hardware implementation of HOL – wherein HOL's inference rules are implemented as machine instructions that modify private memories – and the use of Supervisionary as a general-purpose virtual machine that uses its proof-checking abilities to "challenge" user-space programs to explain their adherence to some system-wide security or correctness policy. Notably, by moving this proof-checking capability into the operating system, or other privileged system software, or even hardware, this capability becomes shared by *all* user-space software executing within the system, not just software written in dedicated "verification aware" programming languages. In a sense, mathematical truth becomes just another resource protected by the operating system and system hardware.

References

1 Oskar Abrahamsson, Magnus O. Myreen, Ramana Kumar, and Thomas Sewell. Candle: A verified implementation of HOL light. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 3:1–3:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.ITP.2022.3`.

[9] `https://www.github.com/DominicPM/supervisionary`
[10] `https://github.com/paritytech/wasmi`
[11] `https://www.wasmtime.dev`

2 Stuart F. Allen, Robert L. Constable, Richard Eaton, Christoph Kreitz, and Lori Lorigo. The Nuprl open logical environment. In David A. McAllester, editor, *Automated Deduction – CADE-17, 17th International Conference on Automated Deduction, Pittsburgh, PA, USA, June 17-20, 2000, Proceedings*, volume 1831 of *Lecture Notes in Computer Science*, pages 170–176. Springer, 2000. `doi:10.1007/10721959_12`.

3 Arm Holdings, Ltd. AArch64 virtual memory system architecture. URL: `https://developer.arm.com/documentation/ddi0406/b/System-Level-Architecture/Virtual-Memory-System-Architecture--VMSA-`, 2023. Accessed 1^{st} May 2023.

4 Andrea Asperti, Wilmer Ricciotti, Claudio Sacerdoti Coen, and Enrico Tassi. The Matita interactive theorem prover. In Nikolaj S. Bjørner and Viorica Sofronie-Stokkermans, editors, *Automated Deduction – CADE-23 – 23rd International Conference on Automated Deduction, Wroclaw, Poland, July 31 – August 5, 2011. Proceedings*, volume 6803 of *Lecture Notes in Computer Science*, pages 64–69. Springer, 2011. `doi:10.1007/978-3-642-22438-6_7`.

5 Edwin C. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *J. Funct. Program.*, 23(5):552–593, 2013. `doi:10.1017/S095679681300018X`.

6 Edwin C. Brady. Idris 2: Quantitative Type Theory in practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming, ECOOP 2021, July 11-17, 2021, Aarhus, Denmark (Virtual Conference)*, volume 194 of *LIPIcs*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.ECOOP.2021.9`.

7 Alonzo Church. A formulation of the Simple Theory of Types. *J. Symb. Log.*, 5(2):56–68, 1940. `doi:10.2307/2266170`.

8 de Ng Dick Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Studies in logic and the foundations of mathematics*, 133:375–388, 1972.

9 D. R. Engler, M. F. Kaashoek, and J. O'Toole. Exokernel: an operating system architecture for application-level resource management. In *SOSP '95: Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266, New York, NY, USA, 1995. ACM. `doi:10.1145/224056.224076`.

10 Michael J. C. Gordon. Introduction to the HOL system. In Myla Archer, Jeffrey J. Joyce, Karl N. Levitt, and Phillip J. Windley, editors, *Proceedings of the 1991 International Workshop on the HOL Theorem Proving System and its Applications, August 1991, Davis, California, USA*, pages 2–3. IEEE Computer Society, 1991.

11 Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979. `doi:10.1007/3-540-09724-4`.

12 Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. Bringing the web up to speed with WebAssembly. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 185–200. ACM, 2017. `doi:10.1145/3062341.3062363`.

13 Norman Hardy. KeyKOS architecture. *ACM SIGOPS Oper. Syst. Rev.*, 19(4):8–25, 1985. `doi:10.1145/858336.858337`.

14 John Harrison. HOL light: An overview. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, volume 5674 of *Lecture Notes in Computer Science*, pages 60–66. Springer, 2009. `doi:10.1007/978-3-642-03359-9_4`.

15 Gérard P. Huet and Hugo Herbelin. 30 years of research and development around Coq. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 249–250. ACM, 2014. `doi:10.1145/2535838.2537848`.

16 Samin S. Ishtiaq and Peter W. O'Hearn. BI as an assertion language for mutable data structures. In Chris Hankin and Dave Schmidt, editors, *Conference Record of POPL 2001: The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, London, UK, January 17-19, 2001*, pages 14–26. ACM, 2001. doi:10.1145/360204.375719.

17 Ralf Jung. *Understanding and evolving the Rust programming language*. PhD thesis, Saarland University, Saarbrücken, Germany, 2020. URL: https://publikationen.sulb.uni-saarland.de/handle/20.500.11880/29647.

18 Stephen Kell, Dominic P. Mulligan, and Peter Sewell. The missing link: explaining ELF static linking, semantically. In Eelco Visser and Yannis Smaragdakis, editors, *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2016, part of SPLASH 2016, Amsterdam, The Netherlands, October 30 – November 4, 2016*, pages 607–623. ACM, 2016. doi:10.1145/2983990.2983996.

19 Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: formal verification of an OS kernel. In Jeanna Neefe Matthews and Thomas E. Anderson, editors, *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 207–220. ACM, 2009. doi:10.1145/1629575.1629596.

20 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *SIGARCH Comput. Archit. News*, 41(1):461–472, March 2013. doi:10.1145/2490301.2451167.

21 Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, pages 461–472, New York, NY, USA, 2013. Association for Computing Machinery. doi:10.1145/2451116.2451167.

22 Robin Milner, Mads Tofte, and Robert Harper. *Definition of Standard ML*. MIT Press, 1990.

23 Georg Moser and Richard Zach. The epsilon calculus (tutorial). In Matthias Baaz and Johann A. Makowsky, editors, *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*, volume 2803 of *Lecture Notes in Computer Science*, page 455. Springer, 2003. doi:10.1007/978-3-540-45220-1_36.

24 George C. Necula. Proof-carrying code. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security, 2nd Ed*, pages 984–986. Springer, 2011. doi:10.1007/978-1-4419-5906-5_864.

25 Kyndylan Nienhuis, Alexandre Joannou, Thomas Bauereiss, Anthony C. J. Fox, Michael Roe, Brian Campbell, Matthew Naylor, Robert M. Norton, Simon W. Moore, Peter G. Neumann, Ian Stark, Robert N. M. Watson, and Peter Sewell. Rigorous engineering for hardware security: Formal modelling and proof in the CHERI design and implementation process. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1003–1020. IEEE, 2020. doi:10.1109/SP40000.2020.00055.

26 Ulf Norell. Interactive programming with dependent types. In Greg Morrisett and Tarmo Uustalu, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA – September 25 – 27, 2013*, pages 1–2. ACM, 2013. doi:10.1145/2500365.2500610.

27 Lawrence C. Paulson, Tobias Nipkow, and Makarius Wenzel. From LCF to Isabelle/HOL. *Form. Asp. Comput.*, 31(6):675–698, December 2019. doi:10.1007/s00165-019-00492-1.

28 Robert Pollack. How to believe a machine-checked proof. In *Twenty Five Years of Constructive Type Theory*. Oxford University Press, October 1998.

29 S. A. Rajunas, Norman Hardy, Allen C. Bomberger, William S. Frantz, and Charles R. Landau. Security in KeyKOS™. In *Proceedings of the 1986 IEEE Symposium on Security and Privacy, Oakland, California, USA, April 7-9, 1986*, pages 78–85. IEEE Computer Society, 1986. doi:10.1109/SP.1986.10000.

30 John C. Reynolds. Separation Logic: A logic for shared mutable data structures. In *17th IEEE Symposium on Logic in Computer Science (LICS 2002), 22-25 July 2002, Copenhagen, Denmark, Proceedings*, pages 55–74. IEEE Computer Society, 2002. doi:10.1109/LICS.2002.1029817.

31 Amitabha Sanyal and Uday P. Khedker. Garbage collection techniques. In Y. N. Srikant and Priti Shankar, editors, *The Compiler Design Handbook: Optimizations and Machine Code Generation, Second Edition*, page 6. CRC Press, 2007.

32 Jonathan S. Shapiro and Norman Hardy. EROS: A principle-driven operating system from the ground up. *IEEE Softw.*, 19(1):26–33, 2002. doi:10.1109/52.976938.

33 Konrad Slind and Michael Norrish. A brief overview of HOL4. In Otmane Aït Mohamed, César A. Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics, 21st International Conference, TPHOLs 2008, Montreal, Canada, August 18-21, 2008. Proceedings*, volume 5170 of *Lecture Notes in Computer Science*, pages 28–32. Springer, 2008. doi:10.1007/978-3-540-71067-7_6.

34 Antonis Stampoulis and Zhong Shao. VeriML: typed computation of logical terms inside a language with effects. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 333–344. ACM, 2010. doi:10.1145/1863543.1863591.

35 Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems—design and implementation, 3rd Edition.* Pearson Education, 2006.

Classical Natural Deduction from Truth Tables

Herman Geuvers
Radboud University, Nijmegen, The Netherlands
Technical University Eindhoven, The Netherlands

Tonny Hurkens
Unaffiliated Researcher, Haps, The Netherlands

Abstract

In earlier articles we have introduced *truth table natural deduction* which allows one to extract natural deduction rules for a propositional logic connective from its truth table definition. This works for both intuitionistic logic and classical logic. We have studied the proof theory of the intuitionistic rules in detail, giving rise to a general Kripke semantics and general proof term calculus with reduction rules that are strongly normalizing. In the present paper we study the classical rules and give a term interpretation to classical deductions with reduction rules. As a variation we define a multi-conclusion variant of the natural deduction rules as it simplifies the study of proof term reduction. We show that the reduction is normalizing and gives rise to the sub-formula property. We also compare the logical strength of the classical rules with the intuitionistic ones and we show that if one non-monotone connective is classical, then all connectives become classical.

2012 ACM Subject Classification Theory of computation → Proof theory; Theory of computation → Type theory; Theory of computation → Constructive mathematics; Theory of computation → Functional constructs

Keywords and phrases Natural deduction, classical proposition logic, multiple conclusion natural deduction, proof terms, formulas-as-types, proof normalization, subformula property, Curry-Howard isomorphism

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.2

Acknowledgements We want to thank the reviewers for their valuable comments.

1 Introduction

Classically, the meaning of a propositional connective is fixed by its *truth table*. This immediately implies consistency, a decision procedure, completeness (with respect to Boolean algebras) for classical logic. Constructively, following the *Brouwer-Heyting-Kolmogorov* interpretation [17], the meaning of a connective is fixed by explaining what a *proof* is that involves the connective. Basically, this explains the *introduction rule(s)* for each connective, from which the elimination rules follow. This was first phrased like this by Prawitz in [14], who studied *natural deduction* in detail, including the reduction of proofs (deductions). By analyzing constructive proofs we then also get consistency (from proof normalization), a decision procedure (from the sub-formula property) and completeness (with respect to Heyting algebras and Kripke models).

In previous papers [6, 7], we have defined a general method to derive natural deduction rules for a connective from its truth table definition, which we have coined TT-ND, Truth Table Natural Deduction. This also works for constructive logic, which we have shown in detail by relating the method to Kripke semantics and by studying proof normalization. For classical logic, a similar method has been described by Milne [10]. The advantage is that the derived rules give natural deduction rules for a connective "in isolation", so without the need to explain a connective in terms of another (e.g. explaining the classical properties of implication using the double negation law). Also, this gives constructive rules for connectives that haven't been studied so far, like if-then-else and nand. These constructive connectives are described and studied in detail in [7]. Finally, it allows to study various properties for a whole set of connectives all at once, like proof normalization and a generic (sound and complete) Kripke semantics. Proof normalization has been defined and studied in [7], where a weak normalization result is proven. Strong normalization has been proven in [8] and [1]. These proofs proceed by defining a proof-term calculus for TT-ND, following the Curry-Howard *proofs-as-terms* (and formulas-as-types) interpretation, and by defining a reduction relation on these proof terms.

These results all apply to the constructive case. In the present paper we study the classical case. We first prove some results only in terms of the logic. We show that for monotonic connectives (like $\vee$, $\wedge$), the classical and constructive rules are equivalent. This has also been shown in [18, 9], but we give a new (arguably simpler) proof. This shows that non-monotonic connectives "make classical logic classical". In our systems, the difference between constructive logic and classical logic for a connective c lies only in the introduction rules for c. To substantiate that non-monotonicity is crucial, we prove that if we allow one classic introduction rule for one non-monotonic connective, all connectives become classical. This implies, for example, that the classical rules for $\to$ imply the (seemingly stronger) classical rules for $\neg$ in presence of the constructive rules for $\neg$.

We also study proof reduction for classical logic derived from truth tables. To do this, we define a proof term calculus which now also has *conclusion variables*, similar in style with $\lambda\mu$ of Parigot [13] or variants of that studied by Ariola and Herbelin [2] and Curien and Herbelin [4]. We define various variants of this, depending on whether one has a single conclusion or multiple conclusions. We also define these as logics and we show – as was to be expected – that multiple conclusion intuitionistic TT-ND is logically equivalent to classical TT-ND. On the proof terms (that include conclusion variables and binding of them), we define a reduction relation that conforms with the reduction of deductions arising from detour elimination and with the goal to obtain a deduction that satisfies the sub-formula property. We describe this in detail for classical multi-conclusion logic, classical single-conclusion logic and intuitionistic single-conclusion logic. We define the reduction on proof-terms, show that it satisfies the subject reduction property and show that proof-terms in normal form satisfy the sub-formula property.

For the study of normalization, we introduce the unified framework of *Truth Table Logic* that arises quite naturally as a system unifying classical/intuitionistic multi-conclusion logic and classical/intuitionistic single conclusion logic. It works with *elimination patterns* and *introduction patterns* which can be combined to form proof terms. For this system we prove strong normalization, and from that, strong normalization for the original intuitionistic logic follows immediately. For the original versions of classical logic, the reduction is actually too "fine-grained" to derive strong normalization directly. But we can conclude that, if there is a proof term in classical logic (multi-conclusion or single conclusion), there is a proof-term in normal form of that same formula. From this we conclude that all logics satisfy the subformula property.

2 Natural Deduction from Truth Tables

We recap our earlier work on Truth Table Natural Deduction. To be able to reason generically about natural deduction rules, all our rules have a "standard form" that looks like this

$$\frac{\Gamma \vdash A_1 \quad \dots \quad \Gamma \vdash A_n \qquad \Gamma, B_1 \vdash D \quad \dots \quad \Gamma, B_m \vdash D}{\Gamma \vdash D}$$

The idea is that, if the conclusion of a rule is $\Gamma \vdash D$, then the hypotheses of the rule can be of one of two forms:

1. $\Gamma \vdash A$: instead of proving D from Γ, we now need to prove A from Γ. We call A a Lemma.
2. $\Gamma, B \vdash D$: we are given extra data B to prove D from Γ. We call B a Casus.

Given this standard form of the rules, we don't have to give the Γ explicitly, as it can be retrieved, so we write

$$\frac{\vdash A_1 \quad \dots \quad \vdash A_n \qquad B_1 \vdash D \quad \dots \quad B_m \vdash D}{\vdash D}$$

Various well-known deduction rules follow this format:

$$\frac{\vdash A \vee B \qquad A \vdash D \qquad B \vdash D}{\vdash D}\vee\text{-el} \qquad \frac{\vdash B}{\vdash A \vee B}\vee\text{-in}_2 \qquad \frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}\wedge\text{-in}$$

But there are others that do not follow this format, for example implication introduction:

$$\frac{A \vdash B}{\vdash A \to B}\to\text{-in}$$

In our set-up, implication introduction will break down in two rules, one introducing the implication, and one discharging the hypothesis. these together are equivalent to standard implication introduction.

$$\frac{A \vdash B}{A \vdash A \to B}\to\text{-in}_1 \qquad \frac{A \vdash A \to B}{\vdash A \to B}\to\text{-in}_2$$

▶ **Definition 1** (Natural Deduction rules from truth tables). *Let c be an n-ary connective c with truth table t_c.*

Each row of t_c gives rise to an elimination rule or an introduction rule for c. (We write $\Phi = c(A_1, \dots, A_n)$.)

$$\begin{array}{ccc|c} A_1 & \dots & A_n & \Phi \\ \hline p_1 & \dots & p_n & 0 \end{array} \quad \mapsto \quad \frac{\vdash \Phi \dots \vdash A_i \ (\textit{if } p_i = 1) \dots A_j \vdash D \ (\textit{if } p_j = 0) \dots}{\vdash D}\,el$$

constructive *intro*

$$\begin{array}{ccc|c} A_1 & \dots & A_n & \Phi \\ \hline q_1 & \dots & q_n & 1 \end{array} \quad \mapsto \quad \frac{\dots \vdash A_i \ (\textit{if } q_i = 1) \dots A_j \vdash \Phi \ (\textit{if } q_j = 0) \dots}{\vdash \Phi}\,in^i$$

classical *intro*

$$\begin{array}{ccc|c} A_1 & \dots & A_n & \Phi \\ \hline r_1 & \dots & r_n & 1 \end{array} \quad \mapsto \quad \frac{\Phi \vdash D \dots \vdash A_i \ (\textit{if } r_i = 1) \dots A_j \vdash D \ (\textit{if } r_j = 0) \dots}{\vdash D}\,in^c$$

We call $\vdash \Phi$ (resp. $\Phi \vdash D$) the major premise *and the other hypotheses of the rule we call the* minor premises*. The minor premises are either a* Lemma*, A_i (if $p_i = 1$ or $q_i = 1$ or $r_i = 1$ in t_c), or a* Casus*, A_j (if $p_j = 0$ or $q_j = 0$ $r_j = 0$) in t_c.*

▶ **Definition 2** (Definition of the logics). *Given a set of connectives $\mathcal{C} := \{c_1, \dots, c_n\}$, we define the* intuitionistic *and* classical *natural deduction systems for $\mathcal{C}$,* IPC$_\mathcal{C}$ *and* CPC$_\mathcal{C}$ *as follows.*

- *Both* IPC$_\mathcal{C}$ *and* CPC$_\mathcal{C}$ *have an* axiom rule

$$\frac{}{\Gamma \vdash A}\,\textit{axiom}(\ \textit{if } A \in \Gamma)$$

- *Both* IPC$_\mathcal{C}$ *and* CPC$_\mathcal{C}$ *have the* elimination rules *for the connectives in $\mathcal{C}$.*

- $\mathsf{IPC}_{\mathcal{C}}$ *has the intuitionistic introduction rules for the connectives in* $\mathcal{C}$*.*
- $\mathsf{CPC}_{\mathcal{C}}$ *has the classical introduction rules for the connectives in* $\mathcal{C}$*.*

In [6], we have given a sound and complete a Kripke semantics for $\mathsf{IPC}_{\mathcal{C}}$. Briefly, a Kripke model is defined as usual and for w a world in the Kripke model, we define $[\![\varphi]\!]_w \in \{0,1\}$ with the meaning that $[\![\varphi]\!]_w = 1$ if and only if formula φ is true in world w. For $\varphi = c(\varphi_1, \ldots, \varphi_n)$, we define $[\![\varphi]\!]_w := 1$ if $t_c([\![\varphi_1]\!]_{w'}, \ldots, [\![\varphi_n]\!]_{w'}) = 1$ for each $w' \geq w$, where t_c is the truth table of c, and otherwise $[\![\varphi]\!]_w := 0$. Similarly, a sound and complete valuation semantics can be given for $\mathsf{CPC}_{\mathcal{C}}$, where a valuation is a map from the proposition letters to $\{0,1\}$ and the interpretation of composite formulas follows the truth table.

▶ **Example 3.** Constructive rules for $\wedge$ (3 elimination rules and one intro rule):

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

$$\dfrac{\vdash A \wedge B \quad A \vdash D \quad B \vdash D}{\vdash D}\wedge\text{-el}_{00} \qquad \dfrac{\vdash A \wedge B \quad A \vdash D \quad \vdash B}{\vdash D}\wedge\text{-el}_{01}$$

$$\dfrac{\vdash A \wedge B \quad \vdash A \quad B \vdash D}{\vdash D}\wedge\text{-el}_{10} \qquad \dfrac{\vdash A \quad \vdash B}{\vdash A \wedge B}\wedge\text{-in}_{11}$$

These rules can be shown to be equivalent to the well-known constructive rules. These rules can be optimized to the three rules we are familiar with.

▶ **Example 4.** Rules for $\neg$: 1 elimination rule and 1 introduction rule.

A	$\neg A$
0	1
1	0

Constructive:

$$\dfrac{\vdash \neg A \quad \vdash A}{\vdash D}\neg\text{-el} \qquad \dfrac{A \vdash \neg A}{\vdash \neg A}\neg\text{-in}^i$$

Classical:

$$\dfrac{\vdash \neg A \quad \vdash A}{\vdash D}\neg\text{-el} \qquad \dfrac{\neg A \vdash D \quad A \vdash D}{\vdash D}\neg\text{-in}^c$$

Using the classical rules for $\neg$, we show that $\neg\neg A \vdash A$ is derivable:

$$\dfrac{\dfrac{\neg\neg A, \neg A \vdash \neg\neg A \quad \neg\neg A, \neg A \vdash \neg A}{\neg\neg A, \neg A \vdash A}\neg\text{-el} \qquad \neg\neg A, A \vdash A}{\neg\neg A \vdash A}\neg\text{-in}^c$$

Simplifying the set of rules

There are various ways to optimize the rules, for example by taking two rules together in one that is equivalent. These have already been described and proven in [6], so we only recap the Lemmas here. To describe these, a first important operation is substituting one derivation on top of another, a kind of "cut operation".

▶ **Lemma 5** (Substituting a deduction in another [6]). *If* $\Gamma \vdash A$ *and* $\Delta, A \vdash B$*, then* $\Gamma, \Delta \vdash B$*.*

Proof. If Σ is a deduction of $\Gamma \vdash A$ and Π is a deduction of $\Delta, A \vdash B$, then we have the following deduction of $\Gamma, \Delta \vdash B$.

$$\begin{array}{c} \vdots\,\Sigma \qquad\qquad \vdots\,\Sigma \\ \Gamma \vdash A \;\;\ldots\;\; \Gamma \vdash A \\ \vdots\,\Pi \\ \Gamma, \Delta \vdash B \end{array}$$

The idea here is that in Π, the leaves that derive $\Delta' \vdash A$ (for some Δ') as an (axiom) is replaced by a copy of the deduction Σ, which is also a deduction of $\Delta', \Gamma \vdash A$ (due to weakening). As contexts only grow if one goes upwards in the tree, we have $\Delta' \supseteq \Delta$, so this new derivation is well-formed. ◀

▶ **Lemma 6** ([6]). *A system with two deduction rules of the following form*

$$\frac{\vdash A_1 \;\ldots\; \vdash A_n \quad B_1 \vdash D \;\ldots\; B_m \vdash D \quad C \vdash D}{\vdash D}$$

$$\frac{\vdash A_1 \;\ldots\; \vdash A_n \quad \vdash C \quad B_1 \vdash D \;\ldots\; B_m \vdash D}{\vdash D}$$

is equivalent to the system with these two rules replaced by

$$\frac{\vdash A_1 \;\ldots\; \vdash A_n \quad B_1 \vdash D \;\ldots\; B_m \vdash D}{\vdash D}$$

We don't repeat the proof, which is in [6], but we give an example.

▶ **Example 7.** The two rules for $\wedge$ from Example 3, ($\wedge$-el$_{00}$) and ($\wedge$-el$_{10}$) can be replaced by one rule:

$$\frac{\vdash A \wedge B \quad B \vdash D}{\vdash D}\,\wedge\text{-el}_{\_0}$$

The intuition is that, as A can occur as a Lemma or as a Casus in an elimination rule where everything else is the same, we can just omit it.

As we can also leave rule ($\wedge$-el$_{00}$) in (as it is derivable), we can do the replacement again, and replace rules ($\wedge$-el$_{00}$) and ($\wedge$-el$_{01}$) by

$$\frac{\vdash A \wedge B \quad A \vdash D}{\vdash D}\,\wedge\text{-el}_{0\_}$$

▶ **Lemma 8.** *[6] A system with a deduction rule of the form to the left is equivalent to the system with this rule replaced by the rule on the right.*

$$\frac{\vdash A_1 \;\ldots\; \vdash A_n \quad B \vdash D}{\vdash D} \qquad\qquad \frac{\vdash A_1 \;\ldots\; \vdash A_n}{\vdash B}$$

Again, we don't repeat the proof, as it is in [6], but we give an example.

▶ **Example 9.** Having the optimized rules for $\wedge$ from Example 7, ($\wedge$-el$_{\_0}$) and ($\wedge$-el$_{0\_}$), we can replaced them by the following rules, which are the well-known elimination rules for $\wedge$.

$$\frac{\vdash A \wedge B}{\vdash B}\,\wedge\text{-el}'_{\_0} \qquad\qquad \frac{\vdash A \wedge B}{\vdash A}\,\wedge\text{-el}'_{0\_}$$

The constructive connectives

We have already seen the $\wedge, \neg$ rules. The optimized rules for $\vee, \to, \top$ and $\bot$ we obtain are:

$$\frac{\vdash A \vee B \quad A \vdash D \quad B \vdash D}{\vdash D}\,\vee\text{-el} \qquad \frac{\vdash A}{\vdash A \vee B}\,\vee\text{-in}_1 \qquad \frac{\vdash B}{\vdash A \vee B}\,\vee\text{-in}_2$$

$$\frac{\vdash A \to B \quad \vdash A}{\vdash B}\to\text{-el} \qquad \frac{\vdash B}{\vdash A \to B}\to\text{-in}_1 \qquad \frac{A \vdash A \to B}{\vdash A \to B}\to\text{-in}_2$$

$$\frac{}{\vdash \top}\,\top\text{-in} \qquad \frac{\vdash \bot}{\vdash D}\,\bot\text{-el}$$

The rules for the classical $\to$ connective

The classical rules for implication are as follows. The elimination is rule is the same as the constructive one and we also have the first introduction rule $\to$-in$_1$. In addition we have the rule on the right. It is classical in the sense that one can derive Peirce's law from it (without using negation).

$$\frac{\vdash A \to B \quad \vdash A}{\vdash B}\to\text{-el} \qquad \frac{\vdash B}{\vdash A \to B}\to\text{-in}_1 \qquad \frac{A \to B \vdash D \quad A \vdash D}{\vdash D}\to\text{-in}_2^c$$

▶ **Example 10.** We give a classical derivation of Peirce's law, using only the classical rules for $\to$.

$$\dfrac{\dfrac{A \vdash A}{A \vdash ((A \to B) \to A) \to A} \qquad \dfrac{\dfrac{\dfrac{(A \to B) \to A \vdash (A \to B) \to A \quad A \to B \vdash A \to B}{A \to B, (A \to B) \to A \vdash A}}{A \to B, (A \to B) \to A \vdash ((A \to B) \to A) \to A}}{A \to B \vdash ((A \to B) \to A) \to A}}{\vdash ((A \to B) \to A) \to A}\to\text{-in}_2^c$$

3 Monotone and non-monotone connectives

▶ **Definition 11.** *A connective c is monotone if its truth table t_c is a monotone function from $\{0,1\}^n$ to $\{0,1\}$ with respect to the ordering $0 \leq 1$.*

Of the standard connectives, $\vee$ and $\wedge$ are monotone, while $\neg$, $\to$ and $\leftrightarrow$ are non-monotone. For monotone connectives, the constructive and classical rules are equivalent. (So this holds for $\wedge$, $\vee$.) For the non-monotone connectives like $\to$ and $\neg$, this is not the case, which is well-known, as Example 10 shows. There is an even stronger result: the classical intro rule for the one ($\to$ or $\neg$) implies the classical intro rule for the other ($\neg$ or $\to$). This hold in general: if we add one classical introduction rule for one non-monotone connective, then all non-monotone connectives are classical.

▶ **Proposition 12.** *For c monotone, the classical and constructive derivation rules are equivalent.*

Proof. Let c be a monotone connective, say with an introduction rule derived from the truth table row $r = (r_1, \ldots, r_i, \ldots, r_n|1)$. If $r_i = 0$, then we also have the truth table row $r' = (r_1, \ldots, 1, \ldots, r_n|1)$. That is, we have rows $(r_1, \ldots, 0, \ldots, r_n|1)$ and $(r_1, \ldots, 1, \ldots, r_n|1)$. Now, the first lemma for simplifying the rules (Lemma 6) says that the i-th minor premise is immaterial for the rule. This reasoning applies to every 0-entry in row r, so we can eliminate all 0-s from r and we obtain an equivalent introduction rule without any Casus, which therefore looks like this:

$$\frac{\Phi \vdash D \quad \vdash A_1 \; \ldots \; \vdash A_m}{\vdash D}$$

Now, by the second lemma for simplifying rules, Lemma 8, this rule is equivalent to

$$\frac{\vdash A_1 \; \ldots \; \vdash A_m}{\vdash \Phi}$$

which is the constructive introduction rule for c. So for c monotone, a classical introduction rule is equivalent to a constructive one. ◀

3.1 For non-monotone connectives, one classical introduction suffices

We now show that, if we have a set of connectives $\mathcal{C}$ and $c \in \mathcal{C}$ is non-monotone, then adding one classical introduction rule for c makes the whole logic classical. So, if we add this one classical introduction rule, we can derive the other classical introduction rules for c and also the classical introduction rules for all other connectives d. In case d is monotone, this doesn't add anything, because of Proposition 12. But, for example in the case of $\neg$ and $\to$, which are both non-monotone, this shows that the classical rule for $\to$ can be derived from the classical rule for $\neg$, and vice versa, the classical rule for $\neg$ can be derived from the classical rule for $\to$. The first might be expected, but the second maybe not, as one might have the impression that the "double negation law" is really stronger then the classical rules for $\to$. Theorem 13 below is even more general, as it says that all non-monotonic classical connectives are equally strong.

It should be noted that, when we talk about a non-monotonic connective c, say of arity n, we should actually speak about a "non-monotonic pattern in t_c", which is an index i ($1 \leq i \leq n$) and a sequence $a_1, \ldots, a_{i-1}, 1, a_{i+1}, \ldots, a_n$ such that $t_c(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_n) = 1$ and $t_c(a_1, \ldots, a_{i-1}, 1, a_{i+1}, \ldots, a_n) = 0$. When we say we "add a classical introduction rule for this non-monotone c", we mean to add a classical introduction rule for a line $(a_1, \ldots, a_{i-1}, 0, a_{i+1}, \ldots, a_n)$ that is part of such a pattern. (There may be other lines $\overrightarrow{b}$ in the truth table of c, with $t_c(\overrightarrow{b}) = 0$, that are not part of such a pattern; making the introduction rule for such a $\overrightarrow{b}$ classical leaves the whole system intuitionistic.) We will not use the terminology of "non-monotonic pattern in t_c", to keep the text simple, and for $\neg$, $\to$ etcetera it isn't relevant. But in the proof we will start from such a situation.

▶ **Theorem 13.** *Let $\mathcal{C}$ be a set of connectives and let $c \in \mathcal{C}$ be non-monotone. If we add one classical introduction rule for c, we can derive the classical rules for all connectives.*

Proof. To simplify the presentation we do not consider the situation where we have optimized rules, but the proof goes through basically unaltered. Let c be an n-ary connective which is not monotone. Then there are rows $r = (r_1 \ldots r_n|1)$ and $s = (s_1 \ldots s_n|0)$ which only differ at position f and such that $r_f = 0$ and $s_f = 1$. So the corresponding classical introduction and elimination rules for c have the same minor premises, except for position f where the introduction rule (based on r) has a Casus while the elimination rule (based on s) has a Lemma.

We want to show that *any* other classical introduction rule, say the one for m-ary connective d based on a row $t = (t_1 \dots t_m|1)$, can be derived from this particular classical introduction rule for c based on row $r = (r_1 \dots r_n|1)$, by just using the constructive introduction rules and the elimination rules of the connectives c and d. Note that c and d need not be different connectives and the language need not contain any other connective.

The problem can be stated more precisely as follows. We have the following rows in the truth tables.

$$\begin{array}{lll} r = (r_1 \dots r_n|1) & \text{for } n\text{-ary } c & r_k \text{ ranges over the 1-entries}, r_\ell \text{ ranges over the 0-entries},\\ s = (s_1 \dots s_n|0) & \text{for } n\text{-ary } c & s_g \text{ ranges over the 1-entries}, s_h \text{ ranges over the 0-entries},\\ t = (t_1 \dots t_m|1) & \text{for } m\text{-ary } d & t_i \text{ ranges over the 1-entries}, t_j \text{ ranges over the 0-entries}. \end{array}$$

Based on these rows, we have the following rules, where D and the As are arbitrary.

$$\frac{c(A_1,\dots,A_n) \vdash D \qquad \vdash A_k \quad (\text{for } r_k = 1) \qquad A_\ell \vdash D \quad (\text{for } r_\ell = 0)}{\vdash D}\, c\text{-intro based on } r$$

$$\frac{\vdash c(A_1,\dots,A_n) \qquad \vdash A_g \quad (\text{for } s_g = 1) \qquad A_h \vdash D \quad (\text{for } s_h = 0)}{\vdash D}\, c\text{-elim based on } s$$

$$\frac{\vdash A_i \quad (\text{for } t_i = 1) \qquad A_j \vdash d(A_1,\dots,A_m) \quad (\text{for } t_j = 0)}{\vdash d(A_1,\dots,A_m)}\, d\text{-intro based on } t, \text{intuitionistic}$$

Fix the formulas $\Phi = d(C_1,\dots,C_m)$ and D (where $C_1,\dots,C_m, D$ are arbitrary). We need to show that the following is derivable (without using classical introduction for d):

$$\frac{\Phi \vdash^1 D \qquad \vdash^2 C_i \quad (\text{for } t_i = 1) \qquad C_j \vdash^3 D \quad (\text{for } t_j = 0)}{\vdash D}$$

We have marked, using $\vdash^1$, $\vdash^2$ and $\vdash^3$, the hypotheses that we will need to derive $\vdash D$ so we can easily refer to them.

Solution from any extra assumption

We first show that we can derive our result D from an arbitrary additional assumption.

▷ Claim 14. For any formula X, we can derive $X \vdash D$ (under the assumptions laid out so far).

Proof of the Claim. Let X be any formula. Define $\Psi = c(B_1,\dots,B_n)$ where:

- $B_u = X$ if $r_u = s_u = 1$,
- $B_u = D$ if $r_u = 0$ and $s_u = 1$ (so this is the case where $u = f$, the only place where rows r and s differ),
- $B_u = \Phi$ if $r_u = s_u = 0$.

We now have the following derivation of $X \vdash D$ (where we still have to create a derivation of the major premise, $X, \Psi \vdash^* D$).

$$\frac{X, \Psi \vdash^* D \qquad X \vdash B_k \quad (r_k = 1) \qquad B_f \vdash D \qquad B_\ell \vdash D \quad (r_\ell = 0)}{X \vdash D}\, c\text{-intro based on } r$$

Note that the minor premises of this rule are all derivable:

- For $r_k = 1$, we have $B_k = X$ and we have $X \vdash X$,
- For $r_\ell = 0$ with $\ell = f$, we have $B_f = D$ and we have $D \vdash D$,
- For $r_\ell = 0$ with $\ell \neq f$, we have $B_\ell = \Phi$ and we have $\Phi \vdash^1 D$, one of our hypotheses.

We now show the derivability of the major premise, $X, \Psi \vdash^* D$, by giving a derivation of $X, \Psi \vdash \Phi$, which, combined with $\Phi \vdash^1 D$, gives $X, \Psi \vdash D$. Here is the derivation:

$$\dfrac{\vdash^2 C_i \text{ (for } t_i = 1) \qquad \dfrac{\Psi \vdash \Psi \quad X, \Psi \vdash B_g \text{ (for } s_g = 1) \quad C_j \vdash B_f \quad B_h \vdash \Phi \text{ (for } s_h = 0)}{X, \Psi, C_j \vdash \Phi \text{ (for } t_j = 0)}\, c\text{-el}}{\dfrac{X, \Psi \vdash \Phi}{X, \Psi \vdash D}\text{ by } \Phi \vdash^1 D}\, d\text{-in}$$

Note that the minor premises of this rule are all derivable:

- For $s_g = 1$ and $r_g = 1$, we have $B_g = X$ and we have $X \vdash X$,
- For $s_g = 1$ with $g = f$, we have $B_f = D$ and we have $C_j \vdash^3 D$, one of our hypotheses,
- For $s_h = 0$, we have $B_h = \Phi$ and we have $\Phi \vdash \Phi$.

So we have shown that, given $\Phi \vdash^1 D$, $\vdash^2 C_i$ (for $t_i = 1$) and $C_j \vdash^3 D$ for $t_j = 0$, we can derive $X \vdash D$ for any formula X. This proves our Claim 14. ◁

Full Solution. If we can find a formula X such that $\vdash X$ then in combination with Claim 14, $X \vdash D$ we get $\vdash D$ so we are done. If we have some standard connectives in our language, then we can take any theorem for X, like $\top$, $\neg\bot$ or $A \to A$. If we happen to be inside a non-empty Γ, we can take X to be any formula in Γ. But in general, we don't have a Γ and we cannot assume any other connectives than c and d (where c and d may even be the same).

If the introduction rule for d that is based on row t has at least one Lemma (C_i), then we can take for X any such C_i. So if in $t = (t_1 \dots t_m|1)$ we have $t_i = 1$ somewhere, we are done.

Now suppose that $t = (0 \dots 0|1)$, so each minor premise is a Casus in the introduction rule for d based on t that we are considering. Let, for A a formula, $\delta(A)$ denote the formula $d(A, \dots, A)$. Then constructive d-intro gives

$$\dfrac{A \vdash \delta(A)}{\vdash \delta(A)}\, d\text{-intro based on } t, \text{intuitionistic}$$

Let's denote by v the row in the truth table for d with just 1s as arguments: either $v = (1 \dots 1|1)$ or $v = (1 \dots 1|0)$.

case $v = (1 \dots 1|1)$.
Then the constructive introduction rule derived from v directly gives $A \vdash \delta(A)$ so we are done: take $X := \delta(A)$ (for arbitrary A), then $\vdash X$ is shown by:

$$\dfrac{\dfrac{A \vdash A}{A \vdash d(A, \dots, A)}\, d\text{-intro based on } v, \text{intuitionistic}}{\vdash d(A, \dots, A)}\, d\text{-intro based on } t, \text{intuitionistic}$$

case $v = (1 \dots 1|0)$.
Now take $X := \delta(\delta(D))$, where D is the formula that we want to prove $\vdash D$ for.

From row v we have an elimination rule for d, in particular we have, for any E:

$$\dfrac{\vdash \delta(D) \qquad \vdash D}{\vdash E}\, d\text{-elim based on } v$$

We take $\delta(\delta(D))$ for E and we have the following derivation of $\delta(\delta(D))$ (and we are done). Note that $\delta(D) \vdash D$ holds by our earlier Claim 14, because $X \vdash D$ for any X.

$$\dfrac{\dfrac{\delta(D) \vdash \delta(D) \qquad \delta(D) \vdash D}{\delta(D) \vdash \delta(\delta(D))}\, d\text{-elim based on } v}{\vdash \delta(\delta(D))}\, d\text{-intro based on } t, \text{intuitionistic}$$

This completes the proof of Theorem 13. ◀

4 Variants of Classical Natural Deduction

There are various ways to move from constructive logic to classical logic. In sequent calculus, this is done by considering multi-conclusion judgments of the form $\Gamma \vdash \Delta$, where Δ is a sequence (or finite set) of formulas, with the intuitive meaning that the conjunction of the formulas in Γ implies the disjunction of the formulas in Δ. We can also make our truth table natural deduction "multi-conclusion", which helps to clarify the connection between the various systems, but more importantly, it is helpful in describing the proof reduction of classical logic. We first introduce these logics and prove their connection.

▶ **Definition 15.** *Let $\mathcal{C}$ be a set of connectives with their associated truth tables and let Γ and Δ be finite sets of formulas over $\mathcal{C}$. We define intuitionistic and classical multi-conclusion logic by considering the following derivation rules.*

$$\frac{}{\Gamma, A \vdash A, \Delta}\ (\mathit{axiom})$$

$$\frac{\Gamma \vdash \Phi, \Delta_0 \ldots \Gamma \vdash A_k, \Delta_k \ldots \quad \ldots \Gamma, A_\ell \vdash \Delta_\ell}{\Gamma \vdash \Delta}\ (\mathit{el})\ \mathit{if}\ \Delta \supseteq \Delta_0, \Delta_k, \Delta_\ell$$

$$\frac{\ldots \Gamma \vdash A_i, \Delta_i \ldots \quad \ldots \Gamma, A_j \vdash \Phi, \Delta_j \ldots}{\Gamma \vdash \Phi, \Delta}\ (\mathit{in\text{-}int})\ \mathit{if}\ \Delta \supseteq \Delta_i, \Delta_j$$

$$\frac{\Gamma, \Phi \vdash \Delta_0 \ldots \Gamma \vdash A_i, \Delta_i \ldots \quad \ldots \Gamma, A_j \vdash \Delta_j \ldots}{\Gamma \vdash \Delta}\ (\mathit{in\text{-}class})\ \mathit{if}\ \Delta \supseteq \Delta_0, \Delta_i, \Delta_j$$

In each rule, $\Phi = c(A_1, \ldots, A_n)$ for some connective c. The A_i and A_j range over the entries with $r_i = 1$ and $r_j = 0$ of some 1-row r for c. The A_k and A_ℓ range over the entries with $r'_k = 1$ and $r'_\ell = 0$ of some 0-row r' for c.

We define the following logics by having the rules (axiom) and (el) and one introduction rule:

1. Int-mc, *intuitionistic multi-conclusion logic, has introduction rule (in-int).*
2. Class-mc, *classical multi-conclusion logic, has introduction rule (in-class).*

For the multi-conclusion logics, we can also let the Δ be a constant in the rules (just like the Γ), as the (axiom) rule gives weakening anyway. We get the original truth table natural deduction rules by restricting these derivation rules to judgments with a single conclusion:

- *(axiom-sc) is rule (axiom) with Δ empty*
- *(el-sc) is rule (el) with Δ a singleton, Δ_0 and the Δ_k empty, and the Δ_ℓ equal to Δ*
- *(in-int-sc) is rule (in-int) with Δ empty*
- *(in-class-sc) is rule (in-class) with Δ a singleton, Δ_0 and the Δ_j equal to Δ, and the Δ_i empty*

We define the original truth table natural deduction logics in this format by having the rules (axiom-sc) and (el-sc) and one introduction rule:

1. Int, *intuitionistic single-conclusion logic, has introduction rule (in-int-sc).*
2. Class, *classical single-conclusion logic, has introduction rule (in-class-sc).*

▶ **Proposition 16.**

1. Int-mc, Class *and* Class-mc *are equivalent in the sense that*

$$\Gamma \vdash \Phi \text{ in Class} \quad \Leftrightarrow \quad \Gamma \vdash \Phi \text{ in Class-mc}$$

$$\Gamma \vdash \Delta \text{ in Int-mc} \quad \Leftrightarrow \quad \Gamma \vdash \Delta \text{ in Class-mc}$$

2. Int *is really weaker than the other 3 systems if $\mathcal{C}$ contains a non-monotone connective.*

Proof.

1. The proof of $\Gamma \vdash \Phi$ in Class $\Leftrightarrow \Gamma \vdash \Phi$ in Class-mc is by showing completeness for both Class and Class-mc in the standard way (using maximally consistent extensions of sets of formulas) with respect to a valuation semantics. For Class-mc, one proves $\Gamma \vdash \Delta \Leftrightarrow \Gamma \models \Delta$, where $\Gamma \models \Delta$ is defined as: for all $v : \mathsf{At} \to \{0, 1\}$, if $\forall \varphi \in \Gamma(v(\varphi) = 1)$, then $\exists \psi \in \Delta(v(\psi) = 1)$. One proves the same result for Class, from which the first equivalence in the Proposition follows.
   For the proof of $\Gamma \vdash \Delta$ in Int-mc $\Leftrightarrow \Gamma \vdash \Delta$ in Class-mc, the $\Rightarrow$ is immediate, while for the $\Leftarrow$ we show that the classical introduction rule (in-class) is derivable in Int-mc: suppose we have $\Gamma, \Phi \vdash \Delta_0$, $\Gamma \vdash A_i, \Delta_i$ (for all appropriate i, according to row r in truth table t_c) and $\Gamma, A_j \vdash \Delta_j$ (for all appropriate j) in Int-mc. We also have $\Gamma, A_j \vdash \Phi, \Delta_j$, so we can apply the intuitionistic introduction rule (in-int) to conclude $\Gamma \vdash \Phi, \Delta$ for $\Delta \supseteq \Delta_i, \Delta_j$. We also have $\Gamma, \Phi \vdash \Delta_0$, so by Substitution Lemma (Lemma 5 extends directly to the case for Int-mc and Class-mc), we conclude $\Gamma \vdash \Delta, \Delta_0$ (for $\Delta \supseteq \Delta_i, \Delta_j$) and we are done.
2. For well-known non-monotone connectives like $\neg$ and $\to$, it is known that Int is really weaker than Class, e.g. one can prove Peirce's law and $\neg\neg A \vdash A$ in Class, which cannot be proven in Int, as one can show by constructing a Kripke counter-model. This also implies that for other non-monotone connective, say c of arity n, the classical rules are really stronger: with the classical rules for c and the intuitionistic rules for $\neg$ one can prove $\neg\neg A \vdash A$. (This follows directly from Theorem 13.) With the intuitionistic rules for c and $\neg$ one cannot prove $\neg\neg A \vdash A$, as the Kripke semantics for intuitionistic truth table natural deduction shows, see [6]. ◀

5 Proof terms for natural deduction

In earlier articles, we gave proof terms for natural deductions. This simplified the study of proof normalization. We followed the Curry-Howard formulas-as-types (and proofs-as-terms) paradigm. We defined systems with judgments $\Gamma \vdash t : B$, where B is a formula, Γ is a set of declarations $\{x_1 : A_1, \ldots, x_n : A_n\}$, where the A_i are formulas and the x_i are term variables such that every x_i occurs at most once in Γ, and t is a proof term. In these systems, the type of a proof term t is the conclusion B of the proof and the types of the free variables x_i of t are the assumptions A_i of the proof. A Lemma D (a sub-proof of formula D as part of the proof of B) is represented by a sub-term of t of type D. A Casus C (a part of the proof that uses C as extra assumption) is represented by a λ-abstraction over a variable of type C.

In this section, we first define proof term calculi for the logics we have already discussed, and also for some logics we have studied in earlier work, but now with both variables for assumptions (hypotheses) and for conclusions (goals).

By using (abstractions over) conclusion variables, we can distinguish between terms like x (representing an assumption A), $\gamma \cdot x$ (representing a proof of $A \vdash A$) and $\mu\gamma : A.\gamma \cdot x$ (representing a Lemma A). It is also very useful to represent multi-conclusion deductions.

5.1 Proof terms for truth table natural deduction with conclusion variables

We define proof term calculi for TT-ND with conclusion variables for various logics:

1. Int, intuitionistic logic,
2. Class, classical logic,
3. Int-mc, intuitionistic multi-conclusion logic,
4. Class-mc, classical multi-conclusion logic,

The judgments of these calculi will be of the form

$$t : (\Gamma \vdash \Delta)$$

where t denotes the proof term, Γ contains the labelled assumptions, typically $\Gamma = x_1, : A_1, \ldots, x_n : A_n$, and Δ contains the labelled conclusions, typically $\Delta = \alpha_1 : B_1, \ldots, \alpha_m : B_m$. Such a judgment expresses that the sequent $A_1, \ldots, A_n \vdash B_1, \ldots, B_m$ is derivable in the corresponding logic. As in sequent calculus, this can informally be read as "t is a proof of the disjunction of the B_j from the conjunction of the A_i". In case we have single conclusion logic Δ is a singleton, but still with a labelled conclusion, so $\Delta = \alpha : B$. We use λ to abstract over assumption variables and μ to abstract over conclusion variables. Each free variable of t is declared in Γ or Δ. Each proof term consists of two parts (separated by $\cdot$). Conclusion variables only occur as left part of a (nested) proof term and assumption variables only as right part.

▶ **Definition 17.** *Let $\mathcal{C}$ be a set of connectives with their associated truth tables and let Γ be an assumption context and Δ a conclusion context for formulas over $\mathcal{C}$. We consider the following derivation rules for terms, where the syntactic class of terms is as follows.*

$$\begin{aligned} t \quad ::= \quad & \alpha \cdot x \mid (\mu\alpha : A.t) \cdot [\overline{\mu\beta : B.t'} \ ; \overline{\lambda z : C.t''}]_r \\ & \mid (\lambda x : A.t) \cdot \{\overline{\mu\beta : B.t'} ; \overline{\lambda z : C.t''}\}_r \mid \gamma \cdot \{\overline{\mu\beta : B.t'} ; \overline{\lambda z : C.t''}\}_r \end{aligned}$$

$$\frac{}{\alpha \cdot x : (\Gamma, x : A \vdash \alpha : A, \Delta)} \ (\textit{axiom})$$

$$\frac{t : (\Gamma \vdash \beta : \Phi, \Delta_0) \ldots p_k : (\Gamma \vdash \alpha_k : A_k, \Delta_k) \ldots \quad \ldots q_\ell : (\Gamma, y_\ell : A_\ell \vdash \Delta_\ell)}{(\mu\beta : \Phi.t) \cdot [\overline{\mu\alpha_k : A_k.p_k} \ ; \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'} : (\Gamma \vdash \Delta)} \ (\textit{el})^*$$

$$\frac{\ldots p_i : (\Gamma \vdash \alpha_i : A_i, \Delta_i) \ldots \quad \ldots q_j : (\Gamma, y_j : A_j \vdash \gamma : \Phi, \Delta_j) \ldots}{\gamma \cdot \{\overline{\mu\alpha_i : A_i.p_i} ; \overline{\lambda y_j : A_j.q_j}\}_r : (\Gamma \vdash \gamma : \Phi, \Delta)} \ (\textit{in-int})^{**}$$

$$\frac{t : (\Gamma, x : \Phi \vdash \Delta_0) \ldots p_i : (\Gamma \vdash \alpha_i : A_i, \Delta_i) \ldots \quad \ldots q_j : (\Gamma, y_j : A_j \vdash \Delta_j) \ldots}{(\lambda x : \Phi.t) \cdot \{\overline{\mu\alpha_i : A_i.p_i} ; \overline{\lambda y_j : A_j.q_j}\}_r : (\Gamma \vdash \Delta)} \ (\textit{in-class})^{***}$$

In each rule, $\Phi = c(A_1, \ldots, A_n)$ for some connective c. Each r is a 1-row for connective c, where A_i ranges over the entries of r with $r_i = 1$ and A_j ranges over the entries with $r_j = 0$. Each r' is a 0-row for c and A_k ranges over the entries of r' with $r'_i = 1$ and A_ℓ ranges over the entries with $r'_\ell = 0$.

Just as in the definition of the logics (Definition 15), each term rule has the same side condition on the Δ in the conclusion: it should be a superset of the Δ's in the hypotheses. In particular, this means that * represents $\Delta \supseteq \Delta_0, \Delta_k, \Delta_\ell$ where k and ℓ range over a number of Δ's, ** represents $\Delta \supseteq \Delta_i, \Delta_j$ where i and j range over a number of Δ's, *** represents $\Delta \supseteq \Delta_0, \Delta_i, \Delta_j$ where i and j range over a number of Δ's. We define the multi-conclusion term calculi by having the rules (axiom) and (el) and one introduction rule:

1. Int-mc*, intuitionistic multi-conclusion calculus, has introduction rule (in-int).*
2. Class-mc*, classical multi-conclusion calculus, has introduction rule (in-class).*

Each rule for terms can be restricted to judgments $t : (\Gamma \vdash \Delta)$ in which Δ is a singleton, just like the corresponding derivation rule for sequents in Definition 15. We define the single-conclusion term calculi by having the rules (axiom-sc) and (el-sc) and one introduction rule:

1. Int*, intuitionistic single-conclusion calculus, has introduction rule (in-int-sc).*
2. Class*, classical single-conclusion calculus, has introduction rule (in-class-sc).*

We want to single out specific sub-term patterns that occur in the elimination rule and the introduction rule.

▶ **Definition 18.** *An* intro pattern *is a sub-term of the form* $\{\overline{\mu\alpha_i : A_i.p_i}; \overline{\lambda y_j : A_j.q_j}\}_r$. *An* elim pattern *is a sub-term of the form* $[\overline{\mu\alpha_k : A_k.p_k}\ ;\ \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'}$.

We say that the intro pattern $\{\overline{\mu\alpha_i : A_i.p_i}; \overline{\lambda y_j : A_j.q_j}\}_r$ *is* well-typed *in* $\Gamma \vdash \Delta$*, if for all* i*,* $p_i : (\Gamma \vdash \alpha_i : A_i, \Delta_i)$ *for some* Δ_i *and for all* j*,* $q_j : (\Gamma, y_j : A_j \vdash \gamma : \Phi, \Delta_j)$ *for some* Δ_j *and all* $\Delta_i, \Delta_j \subseteq \Delta$*. (And all this is relative to line* r *in the truth table for* c*, where* $\Phi = c(A_1, \ldots, A_n)$*.)*

We say that the elim pattern $[\overline{\mu\alpha_k : A_k.p_k}\ ;\ \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'}$ *is* well-typed *in* $\Gamma \vdash \Delta$*, if for all* k*,* $p_k : (\Gamma \vdash \alpha_k : A_k, \Delta_k)$ *for some* Δ_k *and for all* ℓ*,* $q_\ell : (\Gamma, y_\ell : A_\ell \vdash \Delta_\ell)$ *for some* Δ_k *and all* $\Delta_k, \Delta_\ell \subseteq \Delta$*. (And all this is relative to line* r' *in the truth table for some* c*.)*

▶ **Example 19.** The derivation rules for negation are as follows in the classical multi-conclusion logic. We omit the Γ and Δ.

$$\frac{t : (x : \neg B \vdash \alpha : D) \qquad q : (y : B \vdash \alpha : D)}{(\lambda x : \neg B.t) \cdot \{; \lambda y : B.q\}_r : (\vdash \alpha : D)}\ \text{in}\neg \qquad \frac{t : (\vdash \alpha : \neg B) \qquad q : (\vdash \beta : B)}{(\mu\alpha : \neg B.t) \cdot [\mu\beta : B.q\ ;\ ]_{r'} : (\vdash)}\ \text{el}\neg$$

As logics, the relation between these calculi has been given in Proposition 16. We can also give the relation between the term calculi.

▶ **Lemma 20.** *The calculus* Int *is a subsystem of* Int-mc *and the calculi* Class *and* Int-mc *are subsystems of* Class-mc*:*

$$\begin{array}{lclcl} \mathsf{Int} & \subseteq & \mathsf{Int\text{-}mc} & \subseteq & \mathsf{Class\text{-}mc} \\ & & \mathsf{Class} & \subseteq & \mathsf{Class\text{-}mc} \end{array}$$

where Int ⊆ Int-mc *and* Class ⊆ Class-mc *are by the identity on terms and* Int-mc ⊆ Class-mc *is by interpreting*

$$[\![\gamma \cdot \{\overline{\mu\alpha_i : A_i.p_i}; \overline{\lambda y_j : A_j.q_j}\}_r]\!] := (\lambda x : \Phi.\gamma \cdot x) \cdot \{\overline{\mu\alpha_i : A_i.[\![p_i]\!]}; \overline{\lambda y_j : A_j.[\![q_j]\!]}\}_r$$

Proof. The proof is by a straightforward induction on the derivation, using the fact that all systems have weakening: if $t : (\Gamma \vdash \Delta)$ and $\Gamma \subseteq \Gamma'$, $\Delta \subseteq \Delta'$, then $t : (\Gamma' \vdash \Delta')$. ◀

▶ **Example 21.** We consider a proof of the double negation law in classical multi-conclusion calculus, given by the proof term $t : (z : \neg\neg A \vdash \alpha : A)$, where

$$t \quad := \quad (\mu\gamma : \neg\neg A.\gamma \cdot z) \cdot [\mu\beta : \neg A.(\lambda y : \neg A.\beta \cdot y) \cdot \{; \lambda x : A.\alpha \cdot x\}_r\ ;\ ]_{r'}$$

Observe that t contains a sub-term $(\lambda y : \neg A.\beta \cdot y) \cdot \{; \lambda x : A.\alpha \cdot x\}_r$, of type $(\vdash \beta : \neg A, \alpha : A)$, with two different free conclusion variables. Such proof terms, having multiple free conclusion variables, can only occur in multiple-conclusion calculus.

We give the derivation of t, using the rules given in Example 19. Except for the last line, we omit the declarations $z : \neg\neg A$ and $\alpha : A$ from the context.

$$\frac{\gamma \cdot z : (\vdash \gamma : \neg\neg A) \qquad \dfrac{\beta \cdot y : (y : \neg A \vdash \beta : \neg A) \quad \alpha \cdot x : (x : A \vdash \beta : \neg A)}{(\lambda y : \neg A.\beta \cdot y) \cdot \{; \lambda x : A.\alpha \cdot x\}_r : (\vdash \beta : \neg A)}\ \text{in}\neg}{(\mu\gamma : \neg\neg A.\gamma \cdot z) \cdot [\mu\beta : \neg A.(\lambda y : \neg A.\beta \cdot y) \cdot \{; \lambda x : A.\alpha \cdot x\}_r\ ;\ ]_{r'} : (z : \neg\neg A \vdash \alpha : A)}\ \text{el}\neg$$

In the right branch of this derivation, we have as conclusion context $\alpha : A, \beta : \neg A$.

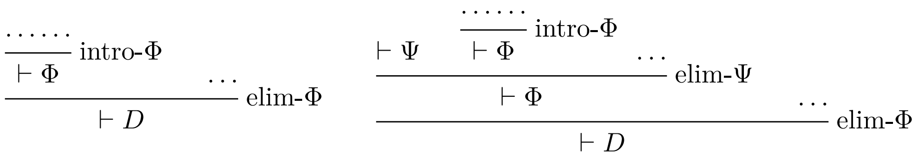


Figure 1 Detour and Permutation.

5.2 Proof terms and reductions: the intuitionistic case

We now describe the term reduction rules. For Int these should correspond to what we have defined and studied in [7] and [8]. There we have defined a proof term calculus for intuitionistic TT-ND (but without conclusion variables), and we have *detour elimination* and *permutation* rules to normalize proofs and to obtain a proof in normal form that satisfies the sub-formula property.

A *detour* in intuitionistic logic is a pattern of an introduction of a formula Φ immediately followed by an elimination of Φ. Such a step can be eliminated by not using Φ at all. This is depicted on the left in Figure 1. A *permutation* is necessary when a detour for Φ is blocked by the elimination of another formula Ψ. Then we first have to permute the two elimination rules, for Φ and Ψ, to make the detour of Φ explicit. See Figure 1 on the right.

It turns out that in our new setting, the conclusion variables nicely take care of the permutation rules. We use the following shorthand notation to improve readability:

- $\overline{p}$ represents $\overline{\mu\alpha_k : A_k.p_k}$,
- $\overline{q}$ represents $\overline{\lambda y_\ell : A_\ell.q_\ell}$.
- $\overline{r}$ represents $\overline{\mu\alpha_i : A_i.r_i}$,
- $\overline{s}$ represents $\overline{\lambda y_j : A_j.s_j}$.

▶ **Definition 22.** *The reduction rule for* Int *is as follows, for $t \neq \beta \cdot x$.*

$$(\mu\beta : \Phi.t) \cdot [\overline{p}\ ;\ \overline{q}]_{r'} \quad \longrightarrow \quad t[\beta := [\overline{p}\ ;\ \overline{q}]_{r'}]$$

Here the substitution $t[\beta := [\overline{p}\ ;\ \overline{q}]_{r'}]$ is defined by, inside t,

$$\begin{array}{llll} \textit{replacing} & \beta \cdot x & \textit{by} & (\mu\alpha : \Phi.\alpha \cdot x) \cdot [\overline{p}\ ;\ \overline{q}]_{r'} & \textit{for a fresh } \alpha \\ \textit{replacing} & \beta \cdot \{\overline{r};\overline{s}\}_r & \textit{by} & q_\ell[y_\ell := \mu\alpha_i : A_i.r_i] & \textit{if } i = \ell \\ & & \textit{or by} & s_j[y_j := \mu\alpha_k : A_k.p_k] & \textit{if } j = k. \end{array}$$

Here $q_\ell[y_\ell := \mu\alpha_i : A_i.r_i]$ is defined by replacing $\gamma \cdot y_\ell$ by $r_i[\alpha_i := \gamma]$.

Note that all free occurrences of β in t should be replaced, including those in s_j. We will see an example in Example 27. Since Int is a single-conclusion calculus, the (only) conclusion variable β of t does not occur in $\mu\alpha_i : A_i.r_i$.

▶ **Lemma 23** (Subject Reduction). *If $t : (\Gamma \vdash \gamma : D)$ and $t \longrightarrow t'$, then $t' : (\Gamma \vdash \gamma : D)$.*

Proof. By induction on the derivation of $t : (\Gamma \vdash \gamma : D)$, which treats the cases where the reduction takes place deeper inside t. The only interesting case is when t is itself a redex. For that we have to prove an auxiliary Substitution Lemma 24, which we give below and from which the case of t itself being a redex follows immediately. ◀

▶ **Lemma 24** (Substitution Property). *Let $\Phi = c(A_1, \ldots, A_n)$ and $t : (\Gamma \vdash \beta : \Phi)$. If the elim pattern $[\overline{\mu\alpha_k : A_k.p_k}\ ;\ \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'}$ is well-typed in $\Gamma \vdash \gamma : D$, where r' is a line in the truth table for c that gives an elimination rule, then $t[\beta := [\overline{p}\ ;\ \overline{q}]_{r'}] : (\Gamma \vdash \gamma : D)$.*

Proof. The proof is by induction on t, using the definition of the substitution given in Definition 22. ◀

The role of the side condition $t \neq \beta \cdot x$ in Definition 22 should be clear: without it we create an infinite reduction sequence right away. As a matter of fact, $(\mu\beta : \Phi.\beta \cdot x) \cdot [\overline{\mu\alpha_k : A_k.p_k}\ ;\ \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'}$ is not a redex as it is an elimination of an assumption, which is exactly what we want from a proof in normal form in Int: that all occurrences of elimination rules eliminate a hypothesis. It is convenient to syntactically characterize the normal forms of Int, $\mathsf{NF}_{\mathsf{Int}}$.

▶ **Definition 25.** *We define* $\mathsf{NF}_{\mathsf{Int}}$*, the normal forms of* Int*, as follows.*

$$\mathsf{NF}_{\mathsf{Int}} ::= \alpha \cdot x \mid (\mu\beta : \Phi.\beta \cdot x) \cdot [\overline{\mu\alpha_k : A_k.p_k}\ ;\ \overline{\lambda y_\ell : A_\ell.q_\ell}]_{r'} \mid \gamma \cdot \{\overline{\mu\alpha_k : A_k.p_k}; \overline{\lambda y_\ell : A_\ell.q_\ell}\}_{r'}$$

where all sub-terms are again in $\mathsf{NF}_{\mathsf{Int}}$.

We give the main properties of normal forms and reduction.

▶ **Lemma 26.**
- $\mathsf{NF}_{\mathsf{Int}}$ *captures precisely the normal forms of* Int.
- *The normal forms of* Int *satisfy the Sub-formula Property: If* $t : (\Gamma \vdash \gamma : D)$, $t \in \mathsf{NF}_{\mathsf{Int}}$*, then for each bound variable,* $\alpha : A$ *or* $x : A$*, occurring in* t*,* A *is a sub-type of a type in* Γ *or* D.
- *The reduction is strongly normalizing.*

Proof. The proof of the first is immediate: terms in $\mathsf{NF}_{\mathsf{Int}}$ cannot reduce and all terms that cannot reduce are in $\mathsf{NF}_{\mathsf{Int}}$. The second follows by induction on the derivation of $t : (\Gamma \vdash \gamma : D)$, where we analyze t based on the cases that arise from $t \in \mathsf{NF}_{\mathsf{Int}}$. The strong normalization property follows from the strong normalization proofs for intuitionistic TT-ND that have been given in [8] and [1]. It also follows from strong normalization of Truth Table Logic (using (2) of Lemma 41) which we prove in Theorem 46. ◀

The term reduction rules of Definition 22 indeed correspond to the ones of [7] and [8]. The interpretation is straightforward, but we do not give it here, because we would have to introduce the definitions of [7] first. To be more precise: every reduction step of the Definition corresponds to a combination of multiple detour elimination/permutation steps from[7]. So the reduction of Definition 22 covers both the detour elimination and the permutation rules. We illustrate this in an example.

▶ **Example 27.** The following deduction has a hidden detour which can be made explicit using a permutation elimination step. (The $\cdots$ are auxiliary parts of the deduction.) The $\to$ -el and $\to$ -in are separated by an $\vee$-el, so the detour arising from an $\to$ -in followed by an $\to$ -el is blocked. The detour can be made explicit using a permutation step. (Note that in TT-ND, the "normal" $\to$-introduction proceeds in two steps, first introducing the $\to$-formula $C \to D$, then abstracting over the hypothesis C.)

$$\dfrac{\dfrac{\dfrac{\cdots}{\vdash A \vee B} \qquad \dfrac{\dfrac{\dfrac{\cdots}{A, C \vdash D}}{A, C \vdash C \to D}\to\text{-in}}{C \to D}\to\text{-in} \qquad \dfrac{\cdots}{B \vdash C \to D}}{C \to D}\vee\text{-el} \qquad \dfrac{\cdots}{\vdash C} \qquad D \vdash D}{D}\to\text{-el}$$

Using the proof terms of [6], this derivation looks like this.

$$\dfrac{\vdash t{:}A \vee B \qquad \dfrac{\dfrac{x{:}A, z{:}C \vdash q{:}D}{x{:}A, z{:}C \vdash \{q \,;\, -\}{:}C \to D}\to\text{-in}}{x{:}A \vdash \{- \,;\, \lambda z{:}C.\{q \,;\, -\}\}{:}C \to D}\to\text{-in} \qquad y{:}B \vdash r{:}C \to D}{\dfrac{\vdash t \bullet [- \,;\, \lambda x{:}A.\{- \,;\, \lambda z{:}C.\{q \,;\, -\}\}, \lambda y{:}B.r]{:}C \to D \qquad\qquad \vdash p{:}C \quad u{:}D \vdash u{:}D}{\vdash t \bullet [- \,;\, \lambda x{:}A.\{- \,;\, \lambda z{:}C.\{q \,;\, -\}\}, \lambda y{:}B.r] \bullet [p \,;\, \lambda u{:}D.u]{:}D}\to\text{-el}}\vee\text{-el}$$

The permutation step for this proof term

$$t \bullet [- \,;\, \lambda x{:}A.\{- \,;\, \lambda z{:}C.\{q \,;\, -\}\}, \lambda y{:}B.r] \bullet [p \,;\, \lambda u{:}D.u]$$

allows one elimination to move into another, in this case to permute the → -el into the ∨-el, leading to

$$t \bullet [- \,;\, \lambda x{:}A.\{- \,;\, \lambda z{:}C.\{q \,;\, -\}\} \bullet [p \,;\, \lambda u{:}D.u], \lambda y{:}B.r \bullet [p \,;\, \lambda u{:}D.u]]$$

Here is the same proof in Int of Definition 17.

$$\dfrac{t{:}(\vdash \alpha{:}A \vee B) \qquad \dfrac{\dfrac{q{:}(x{:}A, z{:}C \vdash \beta{:}D)}{\gamma \bullet \{\mu\beta{:}D.q \,;\, -\}{:}(x{:}A, z{:}C \vdash \gamma{:}C \to D)}}{\gamma \bullet \{- \,;\, \lambda z{:}C.\gamma \bullet \{\mu\beta{:}D.q \,;\, -\}\}{:}(x{:}A \vdash \gamma{:}C \to D)} \qquad r{:}(y{:}B \vdash \gamma{:}C \to D)}{(\mu\alpha{:}A \vee B.t) \bullet [- \,;\, \lambda x{:}A.\gamma \cdot \{- \,;\, \lambda z{:}C.\gamma \bullet \{\mu\beta{:}D.q \,;\, -\}\}, \lambda y{:}B.r]{:}(\vdash \gamma{:}C \to D)}$$

Abbreviating $M := (\mu\alpha{:}A \vee B.t) \bullet [- \,;\, \lambda x{:}A.\gamma \cdot \{- \,;\, \lambda z{:}C.\gamma \bullet \{\mu\beta{:}D.q \,;\, -\}\}, \lambda y{:}B.r]$ we then have

$$\dfrac{M{:}(\vdash \gamma{:}C \to D) \quad p{:}(\vdash \eta{:}C) \quad \delta \bullet u{:}(u{:}D \vdash \delta{:}D)}{(\mu\gamma{:}C \to D.M) \bullet [\mu\eta{:}C.p \,;\, \lambda u{:}D.\delta \bullet u]{:}(\vdash \delta : D)}$$

So the final proof-term is

$$(\mu\gamma{:}C \to D.(\mu\alpha{:}A \vee B.t) \bullet [- \,;\, \lambda x{:}A.\gamma \cdot \{- \,;\, \lambda z{:}C.\gamma \bullet \{\mu\beta{:}D.q \,;\, -\}\}, \lambda y{:}B.r]) \bullet [\mu\eta{:}C.p \,;\, \lambda u{:}D.\delta \bullet u]$$

and the "hidden detour" is directly accessible via the conclusion variable γ. We have underlined the places where a substitution for γ can take place (following Definition 22). It is noteworthy that also the first → -in is contracted with the → -el. If we reduce the term according to Definition 22, we obtain, writing $N := [\mu\eta{:}C.p \,;\, \lambda u{:}D.\delta \bullet u]$

$$(\mu\alpha{:}A \vee B.t) \bullet [- \,;\, \lambda x{:}A.q[\beta := \delta, z := \mu\eta{:}C.p], \lambda y{:}B.r[\gamma := N]]$$

5.3 Proof terms and reductions: the classical multi-conclusion case

We now define reduction for Class-mc. We again use the following shorthand notation to improve readability:

- $\overline{p}$ represents $\overline{\mu\alpha_k : A_k.p_k}$,
- $\overline{q}$ represents $\overline{\lambda y_\ell : A_\ell.q_\ell}$.
- $\overline{r}$ represents $\overline{\mu\alpha_i : A_i.r_i}$,
- $\overline{s}$ represents $\overline{\lambda y_j : A_j.s_j}$.

▶ **Definition 28.** *The reduction rules for* Class-mc *are as follows.*

$$\begin{array}{rcll}(\lambda x : \Phi.t) \cdot \{\overline{r}; \overline{s}\}_r & \longrightarrow & t & \textit{if } x \notin t\\ (\mu\beta : \Phi.t) \cdot [\overline{p}\,;\,\overline{q}]_{r'} & \longrightarrow & t & \textit{if } \beta \notin t\end{array}$$

$$\begin{array}{rl}(\lambda x : \Phi. \ldots (\mu\beta : \Phi. \ldots \beta \cdot x \ldots) \cdot [\overline{p}\,;\,\overline{q}]_{r'} \ldots) \cdot \{\overline{r}; \overline{s}\}_r & \longrightarrow\\ (\lambda x : \Phi. \ldots (\mu\beta : \Phi. \ldots M \ldots) \cdot [\overline{p}\,;\,\overline{q}]_{r'} \ldots) \cdot \{\overline{r}; \overline{s}\}_r & \end{array}$$

$$\begin{array}{rl}(\mu\beta : \Phi. \ldots (\lambda x : \Phi. \ldots \beta \cdot x \ldots) \cdot \{\overline{r}; \overline{s}\}_r \ldots) \cdot [\overline{p}\,;\,\overline{q}]_{r'} & \longrightarrow\\ (\mu\beta : \Phi. \ldots (\lambda x : \Phi. \ldots M \ldots) \cdot \{\overline{r}; \overline{s}\}_r \ldots) \cdot [\overline{p}\,;\,\overline{q}]_{r'} & \end{array}$$

where for M we can choose

$$\begin{array}{rcl}M & = & q_\ell[y_\ell := \mu\alpha_i : A_i.r_i] \textit{ if } i = \ell\\ & = & s_j[y_j := \mu\alpha_k : A_k.p_k] \textit{ if } j = k.\end{array}$$

Here $q_\ell[y_\ell := \mu\alpha_i : A_i.r_i]$ is defined by replacing $\gamma \cdot y_\ell$ by $r_i[\alpha_i := \gamma]$.

Similar to the intuitionistic case (Lemma 23), we have Subject Reduction, which is (again) based on the type soundness of the substitution that is involved.

▶ **Lemma 29** (Subject Reduction). *If $t : (\Gamma \vdash \Delta)$ and $t \longrightarrow t'$, then $t' : (\Gamma \vdash \Delta)$.*

Proof. By induction on the derivation of $t : (\Gamma \vdash \Delta)$, which treats the cases where the reduction takes place deeper inside t. The only interesting case is when t is itself a redex. For that we have to verify that the substitution, of M for $\beta \cdot x$ (and the substitutions that are part of the definition of M) are type correct. That is the case, as we always apply goal variables to assumption variables of the right type, and we substitute expressions under a binder, which avoids the risk of variables being no longer in scope. (And the usual variable hygiene, with renaming of bound variables, avoids capture of free variables by a binder.) ◀

Just as for the single conclusion intuitionistic case, we give an explicit syntax for the normal forms, and give the basic properties for reduction and normal forms, as in Definitions 25 and Lemma 26 for the intuitionistic case.

▶ **Definition 30.** *We define* $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$*, the normal forms of* Class-mc*, as follows.*

$$\mathsf{NF}_{\mathsf{Class\text{-}mc}} ::= \alpha \cdot x \mid (\mu\beta : \Phi.t) \cdot [\overline{p}\,;\,\overline{q}]_{r'} \mid (\lambda x : \Phi.q) \cdot \{\overline{p}; \overline{q}\}_{r'}$$

where all sub-terms are again in $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$ *and*

- *x occurs in q only as $\alpha \cdot x$ with α free in q, and x occurs at least once.*
- *β occurs in t only as $\beta \cdot z$ with z free in t, and β occurs at least once.*

We give the main properties of normal forms and reduction.

▶ **Lemma 31.**
- $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$ *captures precisely the normal forms of* Class-mc.
- *The normal forms of* Class-mc *satisfy the Sub-formula Property: If $t : (\Gamma \vdash \Delta)$, $t \in$* $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$*, then for each bound variable, $\alpha : A$ or $x : A$, occurring in t, A is a sub-type of a type in Γ or Δ.*
- *The calculus is normalizing, in the sense that, if $t : (\Gamma \vdash \Delta)$, then there is a term in normal form q, such that $q : (\Gamma \vdash \Delta)$.*

Proof. The proof of the first is immediate: terms in $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$ cannot reduce and all terms that cannot reduce are in $\mathsf{NF}_{\mathsf{Class\text{-}mc}}$. The second follows by induction on the derivation of $t : (\Gamma \vdash \Delta)$, where we analyze t based on the cases that arise from $t \in \mathsf{NF}_{\mathsf{Class\text{-}mc}}$. The normalization property follows from the strong normalization proof for Truth Table Logic (Theorem 46), which is a generalization of all these systems, and the fact that normal forms in Truth Table Logic can be reflected back to normal forms in Class-mc (Lemma 41 part (3)). ◀

5.4 Proof terms and reductions: the classical single-conclusion case

The reductions for proof-terms in the classical single-conclusion case are a bit less well-behaved than the ones for the multi-conclusion case, because in the classical single conclusion case, there should always be at most one free conclusion variable in a proof term. This means that if t is a proof-term in classical single conclusion logic, then

- a sub-expression $\lambda y : A.q$ of t should have at most one free conclusion variable,
- in a sub-expression $\mu\alpha : A.q$ of t, q should have only α as free conclusion variable.

If one starts from a proof-term t having these properties, and one performs the reductions of Definition 28, one easily ends up in a proof term t' that violates them. Then t' is still valid in multi-conclusion logic, but not in the single conclusion system. The way to prevent this is to first reduce a term to a *lemma-normal form*, which is a term where all Lemmas are assumptions (variables). Those can then be reduced safely in the “standard” way of Definition 28.

This means that reduction for proof terms in single conclusion classical logic proceeds as follows.

- First perform *permutation reductions* to obtain a *lemma-normal form.*
- Then perform *detour reductions*, where a *detour* is an elimination of Φ followed by an introduction of Φ.

This is similar to the constructive case, except for now a term is in “permutation normal form” if all lemmas are axioms.

▶ **Definition 32.** *This is the abstract syntax* $\mathsf{NF}_{\mathsf{lemma}}$ *for* lemma-normal forms*:*

$$\alpha \cdot x \mid (\lambda x{:}A.t) \cdot \{\overline{\mu\alpha_k{:}A_k.\alpha_k \cdot z_k}; \overline{\lambda y_\ell{:}A_\ell.q_\ell}\} \mid (\mu\beta{:}A.\beta \cdot y) \cdot [\overline{\mu\alpha_k{:}A_k.\alpha_k \cdot z_k}\ ;\ \overline{\lambda y_\ell{:}A_\ell.q_\ell}],$$

where x, y, z range over variables and t and the q are again in $\mathsf{NF}_{\mathsf{lemma}}$.

We can obtain a proof term in lemma-normal form by moving applications of an elimination or introduction rule that have a non-trivial Lemma upwards, until all Lemmas become trivial: the proof terms are variables. (Note that $\mu\beta{:}A.\beta \cdot y$ is basically the assumption $y : A$.) This only works in classical logic. If one tries this for intuitionistic proofs, which –from the point of view of classical logic– are proofs with a trivial main premise in the classical introduction rule, one immediately ends up with a proof that has a non-trivial main premise in the classical introduction rule.

We can now specialize Definition 28 to the single-conclusion case by considering only terms in lemma-normal form. We use a similar abbreviation style as before:

- $\overline{z}$ represents $\overline{\mu\alpha_k : A_k.\alpha_k \cdot z_k}$,
- $\overline{q}$ represents $\overline{\lambda y_\ell : A_\ell.q_\ell}$.
- $\overline{v}$ represents $\overline{\mu\alpha_i : A_i.\alpha_i \cdot v_i}$,
- $\overline{s}$ represents $\overline{\lambda y_j : A_j.s_j}$.

▶ **Definition 33.** *In* Class*, the notion of detour and the reduction rules are modified as follows.*

1. *A* detour *is a pattern of the following shape.*

$$(\lambda x : \Phi. \ldots (\mu\beta : \Phi.\beta \cdot x) \cdot [\overline{z}\ ;\ \overline{q}] \ldots) \cdot \{\overline{v}; \overline{s}\}$$

2. *The reduction rules are*

$$\begin{array}{rcl} (\lambda x : \Phi.t) \cdot \{\overline{r}; \overline{s}\}_r & \longrightarrow & t \textit{ if } x \notin t \\ (\lambda x : \Phi. \ldots (\mu\beta : \Phi.\beta \cdot x) \cdot [\overline{z}\ ;\ \overline{q}] \ldots) \cdot \{\overline{v}; \overline{s}\} & \longrightarrow & (\lambda x : \Phi. \ldots M \ldots) \cdot \{\overline{v}; \overline{s}\} \end{array}$$

where for M we can choose

$$\begin{array}{rcl} M & = & q_\ell[y_\ell := v_i] \textit{ if } i = \ell \\ & = & s_j[y_j := z_k] \textit{ if } j = k. \end{array}$$

Again we have Subject Reduction, which basically follows from the classical multi-conclusion case (Lemma 29). The only thing to verify is that we don't go "out of" the single conclusion fragment.

▶ **Lemma 34** (Subject Reduction). *If $t : (\Gamma \vdash \gamma : D)$ in* Class*, $t \in \mathsf{NF}_{\mathsf{lemma}}$ and $t \longrightarrow t'$, then $t' : (\Gamma \vdash \gamma : D)$ in* Class *and $t' \in \mathsf{NF}_{\mathsf{lemma}}$.*

Proof. We don't have to verify the type of t, as its correctness follows from Lemma 29. To verify that t' is a single-conclusion proof term and $t' \in \mathsf{NF}_{\mathsf{lemma}}$ follows from the fact that we never substitute a term under a μ-binder and for assumption variables, we just substitute other assumption variables. ◀

Again, we give an explicit syntax for the normal forms, and the basic properties for reduction and normal forms, as in Definitions 30 and Lemma 31 for the multi-conclusion classical case.

▶ **Definition 35.** *We define $\mathsf{NF}_{\mathsf{Class\text{-}sc}}$, the normal forms of* Class*, as follows.*

$$\alpha \cdot x \mid (\lambda x{:}A.t) \cdot \{\overline{\mu\alpha_k{:}A_k.\alpha_k \cdot z_k}; \overline{\lambda y_\ell{:}A_\ell.q_\ell}\} \mid (\mu\beta{:}A.\beta \cdot y) \cdot [\overline{\mu\alpha_k{:}A_k.\alpha_k \cdot z_k}\ ;\ \overline{\lambda y_\ell{:}A_\ell.q_\ell}],$$

where t and the q_ℓ are again in $\mathsf{NF}_{\mathsf{Class\text{-}sc}}$ and

- *x occurs in t only as $\alpha \cdot x$ with α free in t, and x occurs at least once.*

We give the main properties of normal forms and reduction.

▶ **Lemma 36.**

- $\mathsf{NF}_{\mathsf{Class\text{-}sc}}$ *captures precisely the normal forms of* Class-sc.
- *The normal forms of* Class-sc *satisfy the Sub-formula Property: If $t : (\Gamma \vdash \gamma : D)$, $t \in \mathsf{NF}_{\mathsf{Class\text{-}sc}}$, then for each bound variable, $\alpha : A$ or $x : A$, occurring in t, A is a sub-type of a type in Γ or D.*
- *The calculus is normalizing, in the sense that, if $t : (\Gamma \vdash D)$, then there is a term in normal form q, such that $q : (\Gamma \vdash D)$.*

Proof. The proof is the same as for Lemma 31. For the second second we analyze t based on the cases that arise from $t \in \mathsf{NF}_{\mathsf{Class\text{-}sc}}$. The normalization property follows from the normalization of Class-mc and the fact that, from a normal form in Class-mc we can construct a normal form in Class-sc by taking the lemma-normal form. ◀

6 Truth Table Logic

We now define Truth Table Logic as a unifying logic and proof term calculus for the various logics we have seen before. The idea of Truth Table Logic is that, for $\Phi = c(A_1, \ldots, A_n)$, we decompose an elimination rule for Φ in two parts:

1. the part to be eliminated, $\mu\beta : \Phi.t$, which is the proof of Φ,
2. the *elim pattern*, $[\overline{\mu\alpha_k : A_k.t_k}\ ;\ \overline{\lambda y_\ell : A_\ell.t_\ell}]_{r'}$, consisting of the Lemmas and Casuses that we eliminate Φ with, which are the proofs of A_k (in case A_k is a Lemma) or the proofs from assumption A_ℓ (in case A_ℓ is a Casus).

These elim patterns have already been introduced in Definition 18, but in Truth Table Logic they will get a "first class status", as expressions that have their own type. Similarly we decompose a classical introduction rule in two parts (and give intro patterns a first class status):

1. the part to be introduced, $\lambda y : \Phi.t$, which is the proof from assumption Φ,
2. the *intro pattern*, $\{\overline{\mu\alpha_i : A_i.t_i}; \overline{\lambda y_j : A_j.t_j}\}_r$, consisting of the Lemmas and Casuses that we intro Φ with, which are the proofs of A_i (in case A_i is a Lemma) or the proofs from assumption A_j (in case A_j is a Casus).

For conciseness, we again adopt the earlier abbreviations, where we let $\overline{p}$ and $\overline{r}$ represent series of μ-abstractions, while $\overline{q}$ and $\overline{s}$ represent series of λ-abstractions:

- $\overline{a}$ and $\overline{b}$ typically represent $\overline{\mu\alpha_k : A_k.t_k}$,
- $\overline{f}$ and $\overline{g}$ typicallty represent $\overline{\lambda y_\ell : A_\ell.t_\ell}$.

We will be able to combine intro patterns and elim patterns directly into a proof, without explicit "interference" of an elimination or introduction rule, so we will e.g. have the following as a proof term:

$$\{\overline{b}; \overline{g}\} \cdot [\overline{a}\ ;\ \overline{f}].$$

To make this work, we introduce a new type o, that can informally be read as the type of proofs. It is the only type in Truth Table Logic that is *not* related to a specific formula. In Truth Table Logic, each proof term t will be of the form $f \cdot a$, where the parts f and a have one of the following syntactical forms:

- a variable x or α
- an abstraction $\lambda x.t$ or $\mu\alpha.t$
- an elim pattern $[\overline{a}\ ;\ \overline{f}]$ or an intro pattern $\{\overline{b}; \overline{g}\}$.

These parts will be treated as typed terms on their own and proof terms are just applications $f \cdot a$ that result in type o. Which applications are allowed depends on the variant of Truth Table Logic: we have three variants, where one can choose for intuitionistic or classical logic and for classical logic the single-conclusion or multiple-conclusion variant.

When we consider the "application" $f \cdot a$, we treat f as function and a as argument, not the other way around. If a is of type T, then f should be of type $T \to o$. We abbreviate the function type $T \to o$ to $\sim T$. The type T can be a formula Φ, but it can also be $\sim\Phi$, and then we would have $a : \sim\Phi$ and $f : \sim\sim\Phi$ to make $f \cdot a : o$. So every type T is related to some formula: it can be Φ itself, or $\sim\Phi$, or $\sim\sim\Phi$ or $\sim\sim\sim\Phi$.

The typing of the μ-abstractions and λ-abstractions is determined by the typing of the variables: if $t : o$ and v is an assumption or conclusion variable of type T, then $\mu v{:}T.t : \sim T$ and $\lambda v{:}T.t : \sim T$.

There are three variants of Truth Table Logic, but we first give the generic definition.

▶ **Definition 37** (Truth Table Logic). *Given a set of connectives $\mathcal{C}$ with their truth tables and the propositional formulas generated from $\mathcal{C}$, we define the following classes of types and pre-terms:*

$$\begin{array}{rcl} T & ::= & \Phi \mid {\sim}T \\ t & ::= & f \cdot a \mid a \cdot f \\ f & ::= & \alpha \mid \lambda x : T.t \mid [\overline{a} \,;\, \overline{f}]_r \\ a & ::= & x \mid \mu\alpha : T.t \mid \{\overline{a}; \overline{f}\}_r \end{array}$$

where Φ ranges over formulas, α ranges over conclusion variables, x ranges over assumption variables, and r ranges over the rules of the connectives.

Contexts are of the form $\Gamma; \Delta$, where Γ consists of declarations of assumption variables, typically $\Gamma = x_1 : T_1, \ldots, x_n : T_n$, and Δ consists of declarations of conclusion variables, typically $\Delta = \alpha_1 : T_1, \ldots, \alpha_m : T_m$.

A typing judgment is of the respective forms $\Gamma; \Delta \vdash t : o$, $\Gamma; \Delta \vdash f : T$, $\Gamma; \Delta \vdash a : T$, for some type T. We have the derivation rules below for deriving a typing judgment.

The allowed form of the type T in each of the rules depends on the variant of Truth Table Logic, and will be specified in Definition 38. In the rules (in-pat) and (el-pat), each T_i and T_k is the type of μ-abstractions *and each T_j and T_ℓ is the* type of λ-abstractions *in the specific variant of Truth Table Logic.*

$$\frac{}{\Gamma; \Delta \vdash x : T}\,(hyp),\ \textit{if } x : T \in \Gamma \qquad \frac{}{\Gamma; \Delta \vdash \alpha : T}\,(conc),\ \textit{if } \alpha : T \in \Delta$$

$$\frac{\Gamma, x : T; \Delta \vdash t : o}{\Gamma; \Delta \vdash \lambda x : T.t : {\sim}T}\,(\lambda) \qquad \frac{\Gamma; \Delta, \alpha : T \vdash t : o}{\Gamma; \Delta \vdash \mu\alpha : T.t : {\sim}T}\,(\mu)$$

$$\frac{\Gamma; \Delta \vdash f : {\sim}T \qquad \Gamma; \Delta \vdash a : T}{\Gamma; \Delta \vdash f \cdot a : o}\,(app_1) \qquad \frac{\Gamma; \Delta \vdash a : {\sim}T \qquad \Gamma; \Delta \vdash f : T}{\Gamma; \Delta \vdash a \cdot f : o}\,(app_2)$$

$$\frac{\ldots \Gamma; \Delta \vdash a_i : T_i \ldots \qquad \ldots \Gamma; \Delta \vdash f_j : T_j \ldots}{\Gamma; \Delta \vdash \{\overline{a_i}; \overline{f_j}\}_r : T}\,(\textit{in-pat})$$

$$\frac{\ldots \Gamma; \Delta \vdash a_k : T_k \ldots \qquad \ldots \Gamma; \Delta \vdash f_\ell : T_\ell \ldots}{\Gamma; \Delta \vdash [\overline{a_k} \,;\, \overline{f_\ell}]_r : T}\,(\textit{el-pat})$$

Here, $\Phi = c(A_1, \ldots, A_n)$ and if r is a 1-row for connective c we have the rule (in-pat), where the T_i are related to the A_i for which $r_i = 1$ and the T_j are related to the A_j for which $r_j = 0$. If r is a 0-row for c we have the rule (el-pat), where the T_k are related to the A_k for which $r_k = 1$ and the T_ℓ are related to the A_ℓ for which $r_\ell = 0$.

We now define how the various logics arise from the definition by specifying what is allowed for T in the various derivation rules. In each variant of Truth Table Logic, conclusion variables α for Φ and elim patterns $[\overline{a} \,;\, \overline{f}]_r$ for Φ are of type ${\sim}\Phi$. This implies that μ-abstractions for Φ, $\mu\alpha : {\sim}\Phi.t$, are of type ${\sim}{\sim}\Phi$ and substituting an elim pattern for a conclusion variable in a proof term is well-typed.

▶ **Definition 38.** *Classical multi-conclusion calculus* Class-mc *is specified by*

$\sim\sim\Phi$	$\sim\Phi$	Φ
$\mu\alpha.t$	$\lambda x.t$	
	α	x
	$[\overline{f}\ ;\ \overline{g}]_r$	$\{\overline{f};\overline{g}\}_r$

Intuitionistic single-conclusion calculus Int *is specified by*

$\sim\sim\sim\Phi$	$\sim\sim\Phi$	$\sim\Phi$	Φ
$\lambda x.t$	$\mu\alpha.t$		
	x	α	
		$[\overline{f}\ ;\ \overline{g}]_r$	$\{\overline{f};\overline{g}\}_r$

Classical single-conclusion calculus, Class-sc *is specified by*

$\sim\sim\Phi$	$\sim\Phi$	Φ
$\mu\alpha.t$	$\lambda x.t$	
	α	x
$\{\overline{f};\overline{g}\}_r$	$[\overline{f}\ ;\ \overline{g}]_r$	

In the Class-mc variant of Truth Table Logic, all proof-terms of the forms $(\mu\alpha.t)\cdot[\overline{f}\ ;\ \overline{g}]_r$ and $(\lambda x.t)\cdot\{\overline{f};\overline{g}\}_r$ are permitted. Note that we have reversed some of the applications, and some proof terms can be reduced further than in the original Class-mc.

▶ **Example 39.** We revisit Example 21, where we saw a classical multi-conclusion proof term of type $z : \neg\neg A \vdash \alpha : A$. We can recast that term in Class-mc in Truth Table Logic, and then we have the following term t of type o in the context $z : \neg\neg A; \alpha : \sim A$.

$$t \quad := \quad (\mu\gamma : \sim\neg\neg A.\gamma\cdot z)\cdot[\mu\beta : \sim\neg A.(\lambda y : \neg A.\beta\cdot y)\cdot\{;\lambda x : A.\alpha\cdot x\}_r\ ;\ ]_{r'}$$

But there is a simpler term, because we can normalize t further. (See below for the reduction rules.) Then we get $z : \neg\neg A; \alpha : \sim A \vdash t' : o$ with

$$t' \quad := \quad [\mu\beta : \sim\neg A.\beta\cdot\{;\lambda x : A.\alpha\cdot x\}_r\ ;\ ]_{r'}\cdot z.$$

The derivation in Truth Table Logicis as follows, where we omit $z : \neg\neg A$ and $\alpha : \sim A$ from the context, except for the conclusion.

$$\dfrac{\dfrac{\beta : \sim\neg A \vdash \beta : \sim\neg A \qquad \dfrac{\dfrac{x : A. \vdash \alpha\cdot x : o}{\vdash \lambda x : A.\alpha\cdot x : \sim A}}{\vdash \{;\lambda x : A.\alpha\cdot x\}_r : \neg A}}{\dfrac{\beta : \sim\neg A \vdash \beta\cdot\{;\lambda x : A.\alpha\cdot x\}_r : o}{\vdash [\mu\beta : \sim\neg A.\beta\cdot\{;\lambda x : A.\alpha\cdot x\}_r\ ;\ ]_{r'} : \sim\neg\neg A}} \qquad \vdash z : \neg\neg A}{z : \neg\neg A; \alpha : \sim A \vdash [\mu\beta : \sim\neg A.\beta\cdot\{;\lambda x : A.\alpha\cdot x\}_r\ ;\ ]_{r'}\cdot z : o}$$

In Int, we have reversed the application $\alpha\cdot x$, but that is merely a syntactic reformulation. The main point of Int is that we do not have $(\lambda x.t)\cdot\{\overline{f};\overline{g}\}_r$, to avoid the classical introduction rule. In Int we only have $\gamma\cdot\{\overline{f};\overline{g}\}_r$, which is exactly the term we had for the intuitionistic introduction rule in Definition17.

The system Class-sc is a subsystem of Class-mc, but we avoid the redex $(\lambda x.t)\cdot\{\overline{f};\overline{g}\}_r$, by reversing the order of application to $\{\overline{f};\overline{g}\}_r\cdot(\lambda x.t)$. See below for the reduction rules, where we will enforce that only an abstraction on the left of an application gives a redex. This avoids the possibility of having multiple free conclusion variables in a proof term.

▶ **Definition 40** (Reduction of proof terms in Truth Table Logic)**.** *In the (intuitionistic) variant in which patterns are applied in the order $[\ldots] \cdot \{\ldots\}$ and abstractions in the order $(\lambda x.s) \cdot (\mu\alpha.t)$, proof terms are reduced to proof terms as follows:*

$$\begin{array}{rcl}
(\lambda x : T.t) \cdot a & \longrightarrow & t[x := a] \\
(\mu\alpha : T.t) \cdot f & \longrightarrow & t[\alpha := f] \\
[\overline{b_k}\ ;\ \overline{g_\ell}]_r \cdot \{\overline{a_i}; \overline{f_j}\}_{r'} & \longrightarrow & g_\ell \cdot a_i \\
 & & \textit{if } i = \ell \textit{ as indexes in } 1, \ldots, n, \textit{ where } \Phi = c(A_1, \ldots, A_n) \\
 & & \ \textit{and } r, r' \textit{ are rules for } \Phi \\
[\overline{b_k}\ ;\ \overline{g_\ell}]_r \cdot \{\overline{a_i}; \overline{f_j}\}_{r'} & \longrightarrow & f_j \cdot b_k \\
 & & \textit{if } j = k \textit{ as indexes in } 1, \ldots, n, \textit{ where } \Phi = c(A_1, \ldots, A_n) \\
 & & \ \textit{and } r, r' \textit{ are rules for } \Phi
\end{array}$$

In the classical variants, the order of the patterns in the redex and/or the order of the parts in the reduct is reversed.

That Truth Table Logic is a unification of the logics (actually the calculi) that we have seen in the previous section can be stated and proven precisely.

▶ **Lemma 41.** *For each of the calculi of the previous section,* Int*,* Class *and* Class-mc*, the obvious interpretation $[\![-]\!]$ of proof terms of the logic as proof terms of Truth Table Logic has the following properties.*

1. *If $t : (\Gamma \vdash \Delta)$ in* Int*,* Class *or* Class-mc*, then $\Gamma; \Delta \vdash [\![t]\!] : o$ in Truth Table Logic.*
2. *If $t \longrightarrow q$ in* Int*, then $[\![t]\!] \longrightarrow^+ [\![q]\!]$, where $\longrightarrow^+$ is the transitive closure of $\longrightarrow$.*
3. *If $\Gamma; \Delta \vdash q : o$ in the classical multi-conclusion variant of Truth Table Logic, and q is in normal form, we can reconstruct a proof term t in* Class-mc *such that $[\![t]\!] = q$ and $t : (\Gamma \vdash \Delta)$ in* Class-mc

Proof. The interpretation is the obvious one, and the proof is a direct check of all the cases. ◀

A proof term $f \cdot a$ is a redex if f is an abstraction or both f and a are patterns. In most variants, for each proof term $f \cdot a$ in normal form, at least one of the parts is a variable. In such a calculus, each normal proof has the sub-formula property: except for the assumptions, conclusions and sub-formulas, no other formulas are used.

The normal forms of type o of the intuitionistic variant of our term calculus are of the following shape:

$$x \cdot \alpha \mid \alpha \cdot \{\overline{f}; \overline{g}\}_r \mid x \cdot [\overline{f}\ ;\ \overline{g}]_r$$

In the other variants, the order of the parts of some of these normal forms are reversed. There are no variants with proof terms $f \cdot a$ in which f is a variable and a an abstraction. But the single-conclusion classical variant has proof terms of the form $\{\ldots\} \cdot (\lambda x.t)$. In this calculus, more proof term reductions are needed to get the sub-formula property.

We now prove Strong normalization of reduction for each variant of Truth Table Logic, showing that every reduction sequence leads to a proof term in normal form. We will define, with induction on the type T of terms t the property "t is hereditarily strong normalizing". Then we prove, by induction on the structure of a term t, that each substitution of hereditarily strong normalizing terms for the free variables of t results in a (hereditarily) strongly normalizing term.

For the rest of this section, the variant of Truth Table Logic is fixed. So the types of the conclusion and assumption variables, λ- and μ-abstractions, elim and intro patterns for a formula Φ are specific choices from $\sim\sim\sim\Phi$, $\sim\sim\Phi$, $\sim\Phi$, and Φ.

To prove strong normalization we need the notion of "hereditarily strongly normalizing".

▶ **Definition 42.** *We say that a term t is* HSN (hereditarily strongly normalizing) *if*

1. *t is* SN*,*
2. *if t is an abstraction $\lambda x : T.e$ or $\mu\alpha : T.e$, then $t \cdot a$ is* SN *for each* HSN *$a : T$,*
3. *if t is a pattern $\{\overline{a}; \overline{f}\}_r$ or $[\overline{a}\ ;\ \overline{f}]_r$, then all $\overline{a}, \overline{f}$ are* HSN*.*

Note that this definition is by induction on the type of term t.

Also note that if t is HSN and $t \longrightarrow t'$, then t' is HSN. This follows from the fact that each redex and each reduct is a proof term:

- if t' is an abstraction $\lambda x.e'$ or $\mu\alpha.e'$, then t must be of the form $\lambda x.e$ or $\mu\alpha.e$ where $e \longrightarrow e'$,
- if t' is a pattern $\{\overline{a'}; \overline{f'}\}_r$ or $[\overline{a'}\ ;\ \overline{f'}]_r$, then t must be a pattern $\{\overline{a}; \overline{f}\}_r$ or $[\overline{a}\ ;\ \overline{f}]_r$ that is the same as t' except that for a single i or j, $a_i \longrightarrow a'_i$ or $f_j \longrightarrow f'_j$.

▶ **Definition 43.** *We consider* substitutions *σ that assign terms to variables in a well-typed way:*

- *for each assumption variable x of type T, $\sigma(x) : T$,*
- *for each conclusion variable α of type T, $\sigma(\alpha) : T$.*

Substitution for variables extends in a straightforward way to all terms, so we write $\sigma(t)$ for the result of substituting $\sigma(v)$ for each free occurrence of v in a term t (after having renamed bound variables in t if needed to avoid capturing of free variables of $\sigma(v)$). Note that t and $\sigma(t)$ are terms of the same type.

A substitution σ is HSN *if term $\sigma(v)$ is* HSN *for each variable v.*

A term t is strongly HSN *if term $\sigma(t)$ is* HSN *for each* HSN *substitution σ.*

Since each variable x or α is HSN, the identity substitution is HSN and so each strongly HSN term is HSN.

As usual, reduction rules are closed under substitution: if $t \longrightarrow t'$ then $\sigma(t) \longrightarrow \sigma(t')$. By definition, $t \longrightarrow t'$ if and only if t has a subterm s (possibly $s = t$ itself) that is a redex with reduct s' and t' is the result of replacing s in t by s'. Since, as usual, each free variable of s' is a free variable of s, there is no danger of unintended capturing of a free variable of s' by a surrounding abstraction inside t. Let σ' be the substitution that is like σ, except that $\sigma'(v) = v$ for each variable v that is bound by an abstraction surrounding the subterm s of t. Then for the corresponding reduction $\sigma(t) \longrightarrow \sigma(t')$, the subterm $\sigma'(s)$ of $\sigma(t)$ is replaced by $\sigma'(s)$.

The usually β-reduction $(\lambda x.t) \cdot a \longrightarrow t[x := a]$ has the special property that for each substitution σ, $\sigma((\lambda x.t) \cdot a) \longrightarrow \sigma'(t)$, where the substitution σ' is like σ except that $\sigma'(x) = \sigma(a)$.

▶ **Lemma 44.** *If proof term e is strongly* HSN*, then the abstractions $\lambda x.e$ and $\mu\alpha.e$ are strongly* HSN*.*

Proof. Let e be a strongly HSN proof term and σ an HSN substitution. We do the case for $\lambda x : T.e$. We have to show that $\sigma(\lambda x.e)$ is HSN. Note that $\sigma(\lambda x.e) = \lambda x.\sigma'(e)$ where substitution σ' is like σ, except that $\sigma'(x) = x$. For (1), we need that $\lambda x.\sigma'(e)$ is SN, which follows from the fact that substitution σ' is HSN and term e is strongly HSN. For (2), let $a : T$ be HSN. We have to show that $(\lambda x.\sigma'(e)) \cdot a$ is SN. Since $\lambda x.\sigma'(e)$ and a are SN, an infinite path from $(\lambda x.\sigma'(e)) \cdot a$ must contract a redex $(\lambda x.e') \cdot a'$ where $\sigma'(e) \longrightarrow^* e'$ and $a \longrightarrow^* a'$.

In Truth Table Logic, this reduct is $e'[x := a']$. Note that $\sigma'(e)[x := a'] \longrightarrow^* e'[x := a']$. Let σ'' be the substitution that is like σ and σ' except that $\sigma''(x) = a'$. Substitution σ'' is HSN since term $a' : T$ is HSN. So proof term $\sigma''(e)$ is SN. Now $\sigma''(e) = \sigma'(e)[x := a'] \longrightarrow^* e'[x := a']$, so the reduct $e'[x := a']$ is SN and cannot create an infinite path either. ◀

Note that the situation is different if we add reduction rules like those for Class-mc in which redex $(\lambda x.e') \cdot a'$ can have reducts of the form $(\lambda x.e'') \cdot a'$ where $e' \not\longrightarrow^* e''$ but $e'[x := a'] \longrightarrow^+ e''[x := a']$. Then $\sigma''(e) \longrightarrow^+ e''[x := a']$, so $e''[x := a']$ is SN. This implies that e'' is SN, so an infinite path from $(\lambda x.e'') \cdot a$ must contract a redex again. This cannot happen infinitely often: an infinite path $(\lambda x.e') \cdot a \longrightarrow (\lambda x.e'') \cdot a \longrightarrow (\lambda x.e''') \cdot a \longrightarrow \dots$ would result in an infinite path $e'[x := a'] \longrightarrow^+ e''[x := a'] \longrightarrow^+ e'''[x := a'] \longrightarrow \dots$.

▶ **Lemma 45.** *Each term t is strongly* HSN*.*

Proof. We prove this by induction on t.

- If t is a variable α or x, then for each HSN substitution σ, $\sigma(t)$ is HSN by definition.
- If t is an abstraction $\lambda x.e$ or $\mu\alpha.e$, then the proof term e is strongly HSN by induction, so t is strongly HSN by Lemma 44.
- If t is a pattern $\{\overline{a}; \overline{f}\}_r$ or $[\overline{a}\,;\,\overline{f}]_r$, then all terms $\overline{a}, \overline{f}$ are strongly HSN by induction. Let σ be an HSN substitution. We have to show that $\sigma(t)$ is an HSN term. For (1), we need that $\sigma(t)$ is SN, which follows from the fact that all terms $\sigma(a_i)$ and $\sigma(f_j)$ are SN. For (3), we need that all all terms $\sigma(a_i)$ and $\sigma(f_j)$ are HSN, which holds by induction.
- If t is a proof term $f \cdot a$, then the terms f and a are strongly HSN by induction. Let σ be an HSN substitution. We have to show that $\sigma(t)$ is an HSN term. Since $\sigma(t) = \sigma(f) \cdot \sigma(a)$, we only need to show (1): proof term $\sigma(f) \cdot \sigma(a)$ is SN. Both $\sigma(f)$ and $\sigma(a)$ are HSN and thus SN. An infinite path from $\sigma(f) \cdot \sigma(a)$ must contract a redex $f' \cdot a'$ where $\sigma(f) \longrightarrow^* f'$ and $\sigma(a) \longrightarrow^* a'$. Now both f' and a' are HSN terms. Since $f' \cdot a'$ is a redex in Truth Table Logic, either f' is an abstraction or both f' and a' are patterns. If f' is an HSN abstraction, then by (1), since a' is HSN, $f' \cdot a'$ is SN. If both f' and a' are HSN patterns for some formula $\Phi = c(A_1, \dots, A_n)$ and the reduct is $f'' \cdot a''$, then f'' and a'' are HSN terms for some subformulas A_i, so f'' and a'' are SN. So an infinite path from $f'' \cdot a''$ must contract a redex $f''' \cdot a'''$ again, where $f'' \longrightarrow^* f'''$ and $a'' \longrightarrow^* a'''$. Since f''' has the same type as f'' (and a''' has the same type as a''), related to formula A_i, this cannot happen infinitely often. ◀

▶ **Theorem 46.** *All proof terms in Truth Table Logic are* SN*.*

Proof. By Lemma 45, each (proof) term is strongly HSN, so HSN, so SN. ◀

7 Conclusion

We have shown a couple of basic results for general classical logic derived from truth tables. Most surprisingly maybe is that one classical connective makes the whole logic classical: it is not possible to combine e.g. a classical implication with a constructive negation, as the negation becomes classical due to the fact that implication is classical. Truth Table Natural Deduction, TT-ND, provides the good setting for studying these properties as it gives generic deduction rules for connectives "in isolation", i.e. without explaining one connective in terms of the other.

Then we have studied the proof theory of classical TT-ND, and we have introduced proof terms for classical deductions that use both assumption variables (hypotheses) and conclusion variables. This has enabled us to study proof normalization, with the aim that proofs in normal form satisfy the sub-formula property. The use of conclusion variables turns out to be useful in general, also for the intuitionistic case, where it enables a reduction rule that unifies detour steps and permutation steps. Conclusion variables also naturally enable multi-conclusion natural deduction. Classical multi-conclusion TT-ND is the most general of these systems, where other systems can be embedded into. For this system we prove strong normalization. Classical multi-conclusion TT-ND also emphasizes that there are basically four term-formers: λ-abstraction, μ-abstraction, intro patterns and elim patterns, where the latter two are derived from the truth table. Based on this we define Truth Table Logic as a unifying system.

8 Future and Related Research

For future work, we see the further study of Truth Table Logic as unifying framework for TT-ND. Truth Table Logic also emphasizes the interpretation of proofs of a negated formula as a continuation, where $\sim\Phi$ denotes the type of continuations over Φ. This relates to the general question of the computational interpretation of classical proofs, which has been studied in various research works, like [13, 2, 4], and it is to be studied how our generic computation rules relate to the concrete ones studied in these papers for implication.

Also, the system TT-ND derives rules from a truth table, but that doesn't cover all possible connectives. E.g. if one defines a constructive connective in terms of other, the truth table one obtains generates constructive rules that are sometimes stronger and sometimes weaker than the constructive formula. For example, if we consider the truth table for $c(A, B) := \neg A \to B$, the constructive rules we derive for c are exactly the ones for $A \vee B$, which is stronger than $\neg A \to B$. On the other hand, if we consider the truth table for $d(A, B) := \neg A \vee B$, the constructive rules we derive for c are exactly the ones for $A \to B$, which is weaker than $\neg A \vee B$.

Olkhovikov and Schroeder-Heister [12] also show examples of this, e.g. for $A \vee (B \to C)$ one can write down the truth table and derive constructive rules, but they are weaker than the constructive formula $A \vee (B \to C)$ (because one basically obtains $B \to (A \vee C)$). The TT-ND derived rules give so called "flat elimination" rules [15], and it is likely that it defines exactly the connectives with flat elimination rules. We conjecture that if $c(A_1, \ldots, A_n)$ is a formula defined in terms of the standard connectives, and we derive constructive rules form the truth table for c, then we get a formula equivalent to $c(A_1, \ldots, A_n)$ if in every subformula (of $c(A_1, \ldots, A_n)$) of the shape $P \vee Q$, $Q \vee P$, $P \to Q$ or $\neg P$, P does not contain negation or implication. Or put differently: in P we only have monotone connectives.

This is related to the general study of elimination rules [11, 19], the notion of higher level rules [16] and "harmony" in logic [15, 5]. It would be interesting to see which class of connectives can be defined using TT-ND, and whether the generic approach can be extended to more connectives, e.g. with higher level elimination rules.

Also, based on generalizing $\lambda\mu$ of Parigot [13], and with a semantic view on dualizing implication Crolard [3] has defined the $-$ connective, which has a constructive interpretation that is different from what we would get from a truth table. The interpretation in Kripke models "looks downward", which our interpretation doesn't do. It would be interesting whether the ideas of Crolard can be generalized to other connectives. The relation between Crolard's work and the work on generalized elimination rules and harmony in logic is also unclear.

Finally, there is the obvious question of how these results extend to predicate logic. We are working on extending the TT-ND ideas to predicate logic and define general rules, both classical and constructive for quantifiers, "in isolation", that is without explaining them in terms of other quantifiers.

References

1 Andreas Abel. On model-theoretic strong normalization for truth-table natural deduction. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy*, volume 188 of *LIPIcs*, pages 1:1–1:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.TYPES.2020.1.

2 Z. Ariola and H. Herbelin. Minimal classical logic and control operators. In *ICALP*, volume 2719 of *LNCS*, pages 871–885. Springer, 2003.

3 T. Crolard. A formulae-as-types interpretation of subtractive logic. *J. Log. Comput.*, 14(4):529–570, 2004.

4 P.-L. Curien and H. Herbelin. The duality of computation. In *ICFP*, pages 233–243, 2000.

5 N Francez and R Dyckhoff. A note on harmony. *Journal of Philosophical Logic*, 41(3):613–628, June 2012. doi:10.1007/s10992-011-9208-0.

6 H Geuvers and T Hurkens. Deriving natural deduction rules from truth tables. In *Logic and Its Applications – 7th Indian Conference, ICLA 2017, Kanpur, India, January 5-7, 2017, Proceedings*, volume 10119 of *Lecture Notes in Computer Science*, pages 123–138, 2017. doi:10.1007/978-3-662-54069-5_10.

7 H Geuvers and T Hurkens. Proof terms for generalized natural deduction. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs, TYPES 2017, May 29-June 1, 2017, Budapest, Hungary*, volume 104 of *LIPIcs*, pages 3:1–3:39. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.TYPES.2017.3.

8 H. Geuvers, I. van der Giessen, and T. Hurkens. Strong normalization for truth table natural deduction. *Fundam. Inform.*, 170(1-3):139–176, 2019.

9 Tomoaki Kawano, Naosuke Matsuda, and Kento Takagi. Effect of the choice of connectives on the relation between classical logic and intuitionistic logic. *Notre Dame Journal of Formal Logic*, 63(2), 2022. doi:10.1215/00294527-2022-0016.

10 Peter Milne. Inversion principles and introduction rules. In Heinrich Wansing, editor, *Dag Prawitz on Proofs and Meaning*, pages 189–224. Springer International Publishing, Cham, 2015. doi:10.1007/978-3-319-11041-7_8.

11 S. Negri and J. von Plato. *Structural Proof Theory*. Cambridge University Press, 2001.

12 Grigory K. Olkhovikov and Peter Schroeder-Heister. On flattening elimination rules. *Rev. Symb. Log.*, 7(1):60–72, 2014. doi:10.1017/S1755020313000385.

13 M. Parigot. $\lambda\mu$-calculus: An algorithmic interpretation of classical natural deduction. In *LPAR*, volume 624 of *LNCS*, pages 190–201. Springer, 1992.

14 D. Prawitz. *Natural deduction: a proof-theoretical study*. Almqvist & Wiksell, 1965.

15 S Read. Harmony and autonomy in classical logic. *Journal of Philosophical Logic*, 29(2):123–154, 2000. doi:10.1023/A:1004787622057.

16 Peter Schroeder-Heister. The calculus of higher-level rules, propositional quantification, and the foundational approach to proof-theoretic harmony. *Stud Logica*, 102(6):1185–1216, 2014. doi:10.1007/s11225-014-9562-3.

17 A.S. Troelstra and D. van Dalen. *Constructivism in Mathematics, volume I*. Number volume 121 in Studies in Logic and the Foundations of Mathematics. Elsevier Science, 1988.

18 I. van der Giessen. Natural deduction derived from truth tables, Master Thesis Mathematics, Radboud University Nijmegen, July 2018. URL: http://www.cs.ru.nl/~herman/PUBS/Masterscriptie_IrisvanderGiessen.pdf.

19 J. von Plato. Natural deduction with general elimination rules. *Arch. Math. Log.*, 40(7):541–567, 2001.

On Dynamic Lifting and Effect Typing in Circuit Description Languages

Andrea Colledan
University of Bologna, Italy
INRIA Sophia Antipolis, France

Ugo Dal Lago
University of Bologna, Italy
INRIA Sophia Antipolis, France

Abstract

In the realm of quantum computing, circuit description languages represent a valid alternative to traditional QRAM-style languages. They indeed allow for finer control over the output circuit, without sacrificing flexibility nor modularity. We introduce a generalization of the paradigmatic lambda-calculus Proto-Quipper-M, which models the core features of the quantum circuit description language `Quipper`. The extension, called Proto-Quipper-K, is meant to capture a very general form of dynamic lifting. This is made possible by the introduction of a rich type and effect system in which not only *computations*, but also the very *types* are effectful. The main results we give for the introduced language are the classic type soundness results, namely subject reduction and progress.

2012 ACM Subject Classification Theory of computation → Operational semantics; Theory of computation → Type theory; Hardware → Quantum computation

Keywords and phrases Circuit-Description Languages, λ-calculus, Dynamic lifting, Type and effect systems

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.3

Related Version *Extended Version*: `https://arxiv.org/abs/2202.07636` [2]

Funding The research leading to these results has received funding from the European Union - NextGenerationEU through the Italian Ministry of University and Research under PNRR - M4C2 - I1.4 Project CN00000013 "National Centre for HPC, Big Data and Quantum Computing".

1 Introduction

Despite the undeniable fact that large-scale, error-free quantum hardware has yet to be built [19], research into programming languages specifically designed to be compiled towards architectures including quantum hardware has taken hold in recent years [17]. Most of the proposals in this sense (see [9, 22, 24] for some surveys) concern languages that either express or can be compiled into some form of *quantum circuit* [15], which can then be executed by quantum hardware. This reflects the need to have tighter control over the kind and amount of quantum resources that programs employ. In this scenario, the idea of considering high-level languages that are specifically designed to *describe* circuits and in which the latter are treated as first-class citizens is particularly appealing.

A typical example of this class of languages is `Quipper` [10, 11], whose underlying design principle is precisely that of enriching a very expressive and powerful functional language like `Haskell` with the possibility of manipulating quantum circuits. In other words, programs do not just build circuits, but also treat them like data, as can be seen in the example from Figure 1. `Quipper`'s meta-theory has been studied in recent years through the introduction of a family of research languages that correspond to suitable `Quipper` fragments and extensions,

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 3; pp. 3:1–3:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

```
alice ::  Qubit -> Qubit -> Circ (Bit,Bit)
alice q a = do
 a <- qnot a ‘controlled‘ q
 q <- hadamard q
 (x,y) <- measure (q,a)
 return (x,y)
```

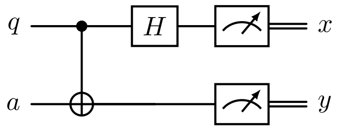


Figure 1 Alice’s part of the quantum teleportation circuit. The `Quipper` program on the left builds the circuit on the right, but while doing so it also manipulates the smaller circuit `qnot a`, enriching it with control.

```
teleport ::  Qubit -> Qubit -> Qubit -> Circ Qubit
teleport b q a = do
 a <- qnot a ‘controlled‘ q
 q <- hadamard q
 (x,y) <- measure (q,a)
 (u,s) <- dynamic_lift (x,y)
 b <- if s then gate_X b else return b
 b <- if u then gate_Z b else return b
 return b
```

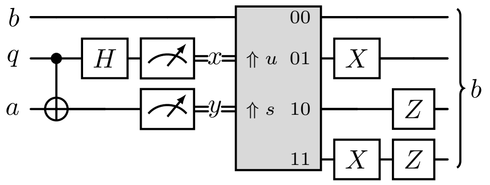


Figure 2 Quantum teleportation circuit with dynamic lifting. The gray box is not a gate, but rather represents the dynamic lifting of bit wires x and y into variables u and s and the extension of the circuit with one of four possible continuations for the remaining wire b, depending on the outcome of the intermediate measurements.

which usually take the form of linear λ-calculi. We are talking about a family of languages whose members include Proto-Quipper-S [21], Proto-Quipper-M [20], Proto-Quipper-D [5, 7], Proto-Quipper-L [13] and Proto-Quipper-Dyn [6].

An aspect that until very recently has only marginally been considered by the research community is the study of the meta-theory of so-called *dynamic lifting*, i.e. the possibility of allowing the (classical) value flowing in one of the wires of the underlying circuit, naturally unknown at circuit building time, to be *visible* in the host program for control flow. As an example, one could append a unitary to some of the wires *only if* a previously performed measurement has yielded a certain outcome. This is commonly achieved in many quantum algorithms via classical control, but Quipper also offers a higher-level solution precisely in the form of dynamic lifting, as can be seen in the example program in Figure 2. Notably, such a program cannot be captured by any of the calculi in the Proto-Quipper family, with the exception of Proto-Quipper-L [13] and Proto-Quipper-Dyn [6], arguably the most recent additions to the family.

Looking at the `Quipper` program in Figure 2, one immediately realizes that the two branches of both occurrences of the `if` operator change the underlying circuit in a uniform way, i.e. the number and type of the wires are the same in either branch. What if, for instance, we wanted to condition the execution of a measurement on a lifted value, like in Figure 3? Unfortunately, `Quipper` does *not* allow the program in Figure 3 to be typed, on account of the two branches of the `if` operators having different types. A program such as this could come from the so-called *measurement calculus* [4], where it would be referred to as a *pattern*, i.e. a sequence of simple operations that act on qubits in a way not so different from what happens in quantum circuits, but with the crucial difference that the basis of a

```
oneWay ::  Qubit -> Qubit -> Bit
oneWay q a = do
 q <- hadamard q
 x <- measure q
 u <- dynamic_lift x
 a <- if u then meas_alpha a else return a
 a <- if not u then meas_beta a else return a
 return a
```

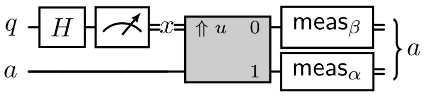


Figure 3 An example of conditional measurement, where meas_α and meas_β measure a qubit in two distinct bases. This program is ill-typed in `Quipper`.

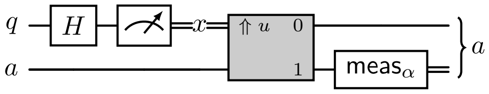


Figure 4 The circuit produced by the program in Figure 3 without the last conditional.

measurement can depend on the outcome of previous measurements. Globally, the program builds a uniformly typed circuit (which always outputs a bit), but locally there are some intermediate steps where the output type is heterogeneous. Of course, in this case, the two `if` operators could be collapsed into one, avoiding local heterogeneity, but there are cases in which keeping two conditionals would be preferable. For example, we might be interested in doing away with the second `if` and reusing the resulting heterogeneous sub-circuit (shown in Figure 4) multiple times when building a more complex circuit, changing only the basis of the second measurement from time to time.

Giving a proper status to the programs that build this kind of heterogeneous circuits would thus endow the programming language with greater flexibility and modularity, so it is natural to wonder whether that of `Quipper` is an intrinsic limitation or if a richer type system can deal with a more general form of circuits.

In this paper, we introduce a generalization of Proto-Quipper-M, called Proto-Quipper-K, in which dynamic lifting is available in a very general form, even more so than in the original `Quipper` language. This newly introduced language is capable of producing not only circuits like the one in Figure 2, but rather a more general class of circuits whose structure and type *essentially depend* on the values flowing through the lifted channels, like the one in Figure 4. We show throughout the paper that this asks for a non-trivial generalization of Proto-Quipper-M's type system, in which types reflect the different behaviors a circuit can have. This is achieved through a type and effect system [16] which assigns any Proto-Quipper-K computation (possibly) distinct types depending on the state of the lifted variables. The main results, beside the introduction of the language itself, its type system, and its operational semantics, are the type soundness results of subject reduction and progress, which together let us conclude that well-typed Proto-Quipper-K programs do not go wrong. An extended version of this paper with more detailed definitions and proofs is available [2].

2 Circuits and Dynamic Lifting: a Bird's-Eye View

This section is meant to provide an informal introduction to the peculiarities of the Proto-Quipper family of paradigmatic programming languages, for the non-specialists. The host language, namely `Haskell`, is modeled as a linearly typed λ-calculus. Terms, in addition

$$\begin{aligned}
&\lambda q_{\mathsf{Qubit}}.\lambda a_{\mathsf{Qubit}}.\mathsf{let}\ (q,a) = \mathsf{apply}(\mathit{CNOT},(q,a))\ \mathsf{in}\\
&\qquad\qquad\qquad \mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \mathsf{apply}(\mathit{Meas2},(q,a))
\end{aligned}$$

Figure 5 A Proto-Quipper-M program describing the circuit shown in Figure 1. We assume that we have a constant boxed circuit $\mathit{CNOT}, H, \mathit{Meas2}$, etc. for every available primitive operation.

to manipulating ordinary data structures and computing (possibly) higher-order functions, are allowed to act on an underlying circuit, which we usually refer to as C. During program evaluation, C can be modified with the addition of wires, gates or entire sub-circuits. This is made possible through a dedicated operator called apply. But how can the programmer specify *where* in C these modifications have to be carried out? This is possible thanks to the presence of *labels*, that is, names that identify distinct output wires of C. These labels are ordinary terms which can be passed around, but have to be treated linearly. Among other things, they can be passed to apply, together with the specification of which gate or sub-circuit is to be appended to the underlying circuit C.

But is apply the only way of manipulating circuits? The answer is negative. Circuits, once built by means of a term, can be "boxed", potentially copied, and passed to other parts of the program, where they can be used, usually by appending them to multiple parts of the underlying circuit. From a linguistic point of view, this is possible thanks to an additional operator, called box, which is responsible for turning a circuit-building term M of type $!(T \multimap U)$ – the type of duplicable functions from label tuples to label tuples – into a term of type $\mathsf{Circ}(T,U)$ – the type of circuits. The term $\mathsf{box}\,M$ does not touch the underlying circuit, but rather evaluates M "behind the scenes", in a sandboxed environment, to obtain a standalone circuit which is then returned as a *boxed circuit*.

Measure as a Label-Lifting Operation

The above considerations are agnostic to the kind of circuits being built. In fact, any type of circuit-like structure can be produced in output by programs of the Proto-Quipper family of languages, provided that it can be interpreted as a morphism in an underlying symmetric monoidal category [20]. If we imagine that the produced structure is an *actual* quantum circuit, however, it is only natural to wonder whether all the examples of circuits that we talked about informally in the introduction can be captured by some instance of Proto-Quipper. Unsurprisingly, the program in Figure 1 is not at all problematic, and can be handled easily by all languages in the Proto-Quipper family (see Figure 5).

The program in Figure 2, on the other hand, can only be handled by Proto-Quipper-L (see Figure 6) and Proto-Quipper-Dyn. In Proto-Quipper-L, a measurement can return the *Boolean value* corresponding to the outcome of the measurement. When this happens, the ongoing computation is split into two branches: one in which the Boolean output is true and one in which it is false. This way, further circuit-altering operations down the line can depend on the classical information produced by the intermediate measurement. In Proto-Quipper-Dyn on the other hand, a bespoke operator dynlift allows to turn a term of type Bit into a term of type Bool, whose result can be used to a similar effect.

How about the program in Figure 3? Unfortunately, this case exceeds the expressiveness of both Proto-Quipper-L and Proto-Quipper-Dyn, in that it requires the distinct execution branches to yield values of different *types*, although only temporarily. The two languages, on the other hand, require all branches of a computation to share the same type, like in `Quipper`. This is where our contribution starts.

$$\begin{array}{l}\lambda b_{\mathsf{Qubit}}.\lambda q_{\mathsf{Qubit}}.\lambda a_{\mathsf{Qubit}}.\mathsf{let}\ (q,a) = (\mathsf{unbox}\ \mathit{CNOT})\,(q,a)\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\ \mathsf{let}\ q = (\mathsf{unbox}\ H)\,q\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\ \mathsf{let}\ (u,s) = (\mathsf{unbox}\ \mathit{MeasLift2})\,(q,a)\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\ \mathsf{let}\ b = \mathsf{if}\ s\ \mathsf{then}\ (\mathsf{unbox}\ X)\,b\ \mathsf{else}\ b\ \mathsf{in}\\ \qquad\qquad\qquad\qquad\ \mathsf{let}\ b = \mathsf{if}\ u\ \mathsf{then}\ (\mathsf{unbox}\ Z)\,b\ \mathsf{else}\ b\ \mathsf{in}\ b\end{array}$$

Figure 6 A Proto-Quipper-L program describing the circuit shown in Figure 2. Informally, terms of the form $(\mathsf{unbox}\ H)\,q$ correspond to terms of the form $\mathsf{apply}(H,q)$ in Proto-Quipper-M.

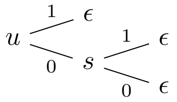


Figure 7 An example of a lifting tree. The empty tree is denoted by ϵ.

The Basic Ideas Underlying Proto-Quipper-K

The approach to dynamic lifting that we follow in this paper is radical. The evaluation of a term M involving the apply operator can give rise to the lifting of a bit value into a variable u and consequently produce in output not a single result in the set VAL of values, but possibly one distinct result for each possible value of u. Therefore, it is natural to think of M as a computation that results in an object in the set $\mathcal{K}_{\{u\}}(\mathit{VAL})$, where $\mathcal{K}_{\{u\}} = X \mapsto (\{u\} \to \{0,1\}) \to X$ is a functor such that, for each possible assignment of a Boolean value to u, $\mathcal{K}_{\{u\}}(X)$ returns an element of X.

What if *more than one* variable is lifted? For example, a program could lift s after having lifted u, but *only if* the latter has value 0. This shows that one cannot just take $\mathcal{K}_{\{u,s\}} = X \mapsto (\{u,s\} \to \{0,1\}) \to X$, simply because not all assignments in $\{u,s\} \to \{0,1\}$ are relevant. Instead, one should just focus on the three assignments $(u=0, s=0), (u=0, s=1)$ and $(u=1)$, namely those assignments which are consistent with the tree in Figure 7, which we call a *lifting tree*. This is a key concept in this work, which we will discuss in detail in Section 3. Our type system captures the lifting pattern of an underlying well-typed program through a lifting tree $\mathfrak{t}$ and the result of the corresponding computation is an element of $\mathcal{K}_{\mathfrak{t}}(\mathit{VAL})$ where $\mathcal{K}_{\mathfrak{t}} = X \mapsto (\mathscr{P}_{\mathfrak{t}} \to X)$ and $\mathscr{P}_{\mathfrak{t}}$ is the set of all assignments of variables which describe a root-to-leaf path in $\mathfrak{t}$. Since by design we want to handle situations in which a circuit, and by necessity the term building it, can produce results which have distinct *types* – and not only distinct *values* – depending on the values of the lifted variables, we also employ (in the spirit of the type and effects paradigm [16]) an effectful notion of *type*, in which computations are typed according to an element of $\mathcal{K}_{\mathfrak{t}}(\mathit{TYPE})$, where TYPE is the set of Proto-Quipper-K types.

3 Generalized Quantum Circuits

A quantum circuit describes a quantum computation by means of the application of *quantum gates*, which represent basic unitary operations, to a number of typed *input wires*, to obtain a number of typed *output wires*. In this section, we introduce a general form of circuits in which the application of *any* gate to one or more wires can be carried out *conditionally* on the classical value flowing in a lifted channel. This is possible even when the gate inputs are *not* the same number and type of the gate outputs. This implies that the number and type of the outputs of a circuit can depend on the values flowing in its wires.

$$\begin{array}{llll} \text{M-types} & \mathit{MTYPE} & T, U & ::= \mathbb{1} \mid w \mid T \otimes U. \\ \text{M-values} & \mathit{MVAL} & \vec{\ell}, \vec{k} & ::= * \mid \ell \mid (\vec{\ell}, \vec{k}). \end{array}$$

$$\text{unit}\,\frac{}{\emptyset \vdash_m * : \mathbb{1}} \qquad \text{label}\,\frac{}{\ell : w \vdash_m \ell : w} \qquad \text{tuple}\,\frac{Q \vdash_m \vec{\ell} : T \quad L \vdash_m \vec{k} : U}{Q, L \vdash_m (\vec{\ell}, \vec{k}) : T \otimes U}$$

Figure 8 Syntax and rule system for M-types and M-values.

A Syntax for Circuits

We represent the inputs and outputs of a circuit as *label contexts*, that is, partial mappings from the set $\mathscr{L}$ of *label names* to the set $\mathscr{W} = \{\mathsf{Bit}, \mathsf{Qubit}\}$ of wire types. The set $\mathscr{L}$ contains precisely the kind of labels that we mentioned in Section 2, therefore a label context is a way to attach type information to labeled wires. We write the set of all label contexts as $\mathscr{Q}$.

Whereas the order of wires in a circuit as a whole is irrelevant, the order of wires in a *gate application* is crucial. For this reason, we perform gate applications not on label contexts, but rather on *label tuples*, which imbue label contexts with a specific ordering via a simple form of typing judgment. The grammar and typing rules for label tuples, which we call *M-values* and whose types we call *M-types*, following [20], are given in Figure 8. Note that $\ell \in \mathscr{L}, w \in \mathscr{W}$, and Q and L are label contexts whose disjoint union is denoted by Q, L.

▶ **Definition 1** (Gate Set). *Let $\mathscr{G}$ be a set of* gates*, equipped with two functions* $\mathsf{inType} : \mathscr{G} \to \mathit{MTYPE}$ *and* $\mathsf{outType} : \mathscr{G} \to \mathit{MTYPE}$. *We denote by $\mathscr{G}(T, U)$ the set of gates with input type T and output type U.*

As an example, $\mathscr{G}(\mathsf{Qubit}, \mathsf{Qubit})$ includes the so-called Hadamard gate, which is used to put a single qubit into a perfect superposition. Besides the set of labels $\mathscr{L}$, there is also another set of names, called $\mathscr{V}$, which is disjoint from it and contains the lifted variables. An *assignment* of lifted variables is then simply a finite sequence of equalities $(u_1 = p_1, \dots, u_n = p_n)$ which assign the values $p_1, \dots, p_n \in \{0, 1\}$ to the distinct variables $u_1, \dots, u_n \in \mathscr{V}$, respectively. We usually indicate assignments with metavariables such as a, b and c.

We now introduce a low-level language to describe quantum circuits at the gate level, which will serve as a target for circuit building in Proto-Quipper-K. We call it *circuit representation language* (CRL) and define it via the following grammar:

$$C, D ::= \mathsf{input}(Q) \mid C; a\,?\,g(\vec{\ell}) \to \vec{k} \mid C; a\,?\,\mathsf{lift}(\ell) \Rightarrow u. \tag{1}$$

The base case $\mathsf{input}(Q)$ corresponds to the trivial identity circuit that takes as input the wires represented by Q and does nothing to them. The notation $a\,?\,g(\vec{\ell}) \to \vec{k}$ denotes the application of a gate g to the wires identified by $\vec{\ell}$ to obtain the wires in $\vec{k}$, provided that the condition expressed by a is met. We simply write $g(\vec{\ell}) \to \vec{k}$ when a gate is applied unconditionally (i.e. when $a = \emptyset$). Figure 9a shows a simple example of a CRL circuit consisting exclusively of gate applications.

On the other hand, $a\,?\,\mathsf{lift}(\ell) \Rightarrow u$ represents the dynamic lifting of the bit wire ℓ if the condition expressed by a is met. When we perform dynamic lifting on a bit, we promote its contents to a Boolean value that is bound to the lifted variable u. This variable can then be mentioned in subsequent assignments to control whether further operations in the circuit are executed or not. The introduction of u thus naturally leads to two distinct execution branches: one in which $u = 0$ and one in which $u = 1$. As we do for gate applications, we write $\mathsf{lift}(\ell) \Rightarrow u$ when a lifting operation is unconditional. A CRL circuit that performs dynamic lifting is shown in Figure 9b.

$\mathsf{input}(q_0 : \mathsf{Qubit}, a_0 : \mathsf{Qubit});$
$\mathsf{CNOT}(q_0, a_0) \rightarrow (q_1, a_1);$
$\mathsf{H}(q_1) \rightarrow q_2;$
$\mathsf{Meas2}(q_2, a_1) \rightarrow (x, y);$

(a) Circuit from Figure 1.

$\mathsf{input}(b_0 : \mathsf{Qubit}, q_0 : \mathsf{Qubit}, a_0 : \mathsf{Qubit});$
$\mathsf{CNOT}(q_0, a_0) \rightarrow (q_1, a_1);$
$\mathsf{H}(q_1) \rightarrow q_2;$
$\mathsf{Meas2}(q_2, a_1) \rightarrow (x, y);$
$\mathsf{lift}(x) \Rightarrow u;$
$\mathsf{lift}(y) \Rightarrow s;$
$(s = 1)\,?\,\mathsf{X}(b_0) \rightarrow b_1;$
$(u = 1; s = 0)\,?\,\mathsf{Z}(b_0) \rightarrow b_2;$
$(u = 1; s = 1)\,?\,\mathsf{Z}(b_1) \rightarrow b_3;$

(b) Circuit from Figure 2.

Figure 9 Two examples of CRL descriptions of a quantum circuit.

Lifting Trees

Naturally, not all CRL expressions denote reasonable circuits. For example, conditioning the application of a gate on the value of a lifted variable which has not yet been introduced should be avoided, for obvious reasons. Capturing this idea at the type level is nontrivial, since the presence of a variable can itself depend on previous liftings. This is where the concept of lifting tree, which we introduced informally in Section 2, really comes into play.

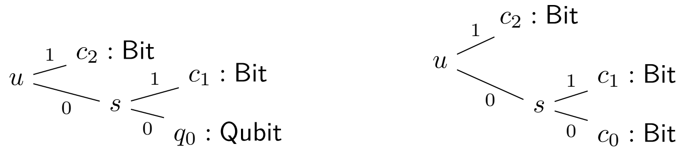
▶ **Definition 2** (Lifting Tree). *We define the set $\mathscr{T}$ of* lifting trees*, along with their* variable set on assignment a*, seen as a function* $\mathsf{var}_a : \mathscr{T} \rightarrow \mathcal{P}(\mathscr{V})$*, as the smallest set of expressions and functions such that*

- $\epsilon \in \mathscr{T}$ *with* $\mathsf{var}_a(\epsilon) = \emptyset$.
- *If* $\mathsf{t}_0 \in \mathscr{T}$ *and* $\mathsf{t}_1 \in \mathscr{T}$*, then for every u that is neither in* $\mathsf{var}_\emptyset(\mathsf{t}_0)$ *nor in* $\mathsf{var}_\emptyset(\mathsf{t}_1)$ *we have* $u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\} \in \mathscr{T}$ *and*

$$\mathsf{var}_a(u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\}) = \{u\} \cup \begin{cases} \mathsf{var}_a\,(\mathsf{t}_0) & \text{if } a(u) = 0, \\ \mathsf{var}_a\,(\mathsf{t}_1) & \text{if } a(u) = 1, \\ \mathsf{var}_a(\mathsf{t}_0) \cup \mathsf{var}_a(\mathsf{t}_1) & \text{if } a(u) \text{ is undefined.} \end{cases} \tag{2}$$

We often write $\mathsf{var}(\mathsf{t})$ as shorthand for $\mathsf{var}_\emptyset(\mathsf{t})$ to denote all the variables mentioned in t. By way of lifting trees, we can keep track of whether an assignment, representing a condition, is consistent with the current state of the lifted variables. Given a lifting tree t, we call $\mathscr{A}_\mathsf{t}$ the set of such assignments (which is easily defined by induction on t, see [2]). Among these consistent assignments, there are some which are *maximal*, i.e. that cannot be further extended: they describe root-to-leaf paths in t and correspond exactly to the elements of the set $\mathscr{P}_\mathsf{t}$ which we introduced in Section 2. Unsurprisingly, for all t we have $\mathscr{P}_\mathsf{t} \subseteq \mathscr{A}_\mathsf{t}$. As an example, let t be the tree from Figure 7. We have $\mathsf{var}(\mathsf{t}) = \{u, s\}, \mathscr{A}_\mathsf{t} = \{\emptyset, (u = 0), (u = 1), (s = 0), (s = 1), (u = 0, s = 0), (u = 0, s = 1)\}$ and $\mathscr{P}_\mathsf{t} = \{(u = 1), (u = 0, s = 0), (u = 0, s = 1)\}$.

Finally, we can formally define one of the key notions of this paper, not only for circuits, but also for programs: given a generic set X, $\mathcal{K}_\mathsf{t}(X)$ indicates the set $\mathscr{P}_\mathsf{t} \rightarrow X$, which we call the *lifting* of X, and whose elements we refer to as *lifted objects*. Despite the fact that



(a) Before composition. **(b)** After composition.

Figure 10 An example of composition with overwriting.

lifted objects are formally mappings, seeing them as decorated lifting trees whose leaves are labeled with objects from X is perhaps more intuitive. Following this interpretation, given $x \in X$, we indicate by $\{x\}$ the trivial lifted object in $\mathcal{K}_\epsilon(X)$ defined as the mapping $\emptyset \mapsto x$, and by $u\,\{\xi_0\}\{\xi_1\}$ the object in $\mathcal{K}_{u\,\{\mathsf{t}_0\}\{\mathsf{t}_1\}}(X)$ defined as $a \mapsto \xi_{a(u)}(a')$, where $\xi_0 \in \mathcal{K}_{\mathsf{t}_0}(X), \xi_1 \in \mathcal{K}_{\mathsf{t}_1}(X)$ and $a' \in \mathscr{P}_{\mathsf{t}_{a(u)}}$ is obtained from a by excluding u from its domain. A graphical representation of this intuition can be found in the form of the trees shown in figures 10 and 11.

▶ **Example 3.** Consider Figure 9. The CRL circuit on the left does not perform lifting and therefore has a trivial output $\{x : \mathsf{Bit}, y : \mathsf{Bit}\}$, while the CRL circuit on the right does and has output $u\,\{s\,\{b_0 : \mathsf{Qubit}\}\{b_1 : \mathsf{Qubit}\}\}\{s\,\{b_2 : \mathsf{Qubit}\}\{b_3 : \mathsf{Qubit}\}\}$.

The same intuition informs the various operations that we define homogeneously on lifting trees and lifted objects. The first is a very natural one: if t is a lifting tree and $\{x_a\}_{a\in I}$ is a family of elements in X indexed on a subset $I \subseteq \mathscr{P}_\mathsf{t}$ of the root-to-leaf paths of t, then the expression $\mathsf{t}[x_a]_a^I$ stands for the lifted object obtained by sticking each x_a to the leaf of t identified by $a \in I$. Note that this operation, which we call *composition*, is generally loosely typed and can give rise to heterogeneous lifted objects. However, in the case $I = \mathscr{P}_\mathsf{t}$, the resulting lifted object belongs to $\mathcal{K}_\mathsf{t}(X)$. In this case we also write $\mathsf{t}[x_a]_a$ as shorthand for $\mathsf{t}[x_a]_a^{\mathscr{P}_\mathsf{t}}$, whereas when I is a singleton $\{b\}$ we usually omit the subscript a and write $\mathsf{t}[x]^{\{b\}}$ for $\mathsf{t}[x_a]_a^{\{b\}}$. We also allow already specified lifted objects to appear on the left of a composition, in which case we overwrite the interested leaves. For instance, if Δ is a lifted label context, we often write the expression $\Delta[Q]^{\{a\}}$ to denote the lifted label context that associates Q to a and is otherwise equal to Δ on all the other branches.

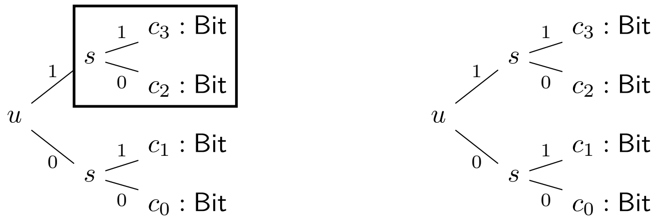
▶ **Definition 4** (Composition). *Given $\xi \in \mathcal{K}_\mathsf{t}(X)$, an index set $I \subseteq \mathscr{P}_\mathsf{t}$ and a family $\{x_a\}_{a\in I}$ of elements in X, we define the* composition of ξ and $\{x_a\}_{a\in I}$, *written $\xi[x_a]_a^I$, as*

$$\begin{aligned} \{y\}[x_a]_a^\emptyset &= \{y\}, \\ \{y\}[x_a]_a^{\{\emptyset\}} &= \{x_a\}, \\ u\,\{\xi_0\}\{\xi_1\}[x_a]_a^I &= u\,\{\xi_0[x_a]_a^{I_0}\}\{\xi_1[x_a]_a^{I_1}\}, \end{aligned} \tag{3}$$

where $I_b = \{a|_{\mathsf{var}(\mathsf{t})\setminus\{u\}} \mid a \in I \wedge a(u) = b\}$ and $a|_{\mathsf{var}(\mathsf{t})\setminus\{u\}}$ denotes the exclusion of u from the domain of a.

▶ **Example 5.** Let $\mathsf{t} = u\,\{s\,\{\epsilon\}\{\epsilon\}\}\{\epsilon\}$ and let $\Delta = u\,\{s\,\{q_0 : \mathsf{Qubit}\}\{c_1 : \mathsf{Bit}\}\}\{c_2 : \mathsf{Bit}\} \in \mathcal{K}_\mathsf{t}(\mathscr{Q})$ be a lifted label context, which graphically corresponds to the tree shown in Figure 10a. We have $\Delta[c_0 : \mathsf{Bit}]^{\{u=0,s=0\}} = u\,\{s\,\{c_0 : \mathsf{Bit}\}\{c_1 : \mathsf{Bit}\}\}\{c_2 : \mathsf{Bit}\} \in \mathcal{K}_\mathsf{t}(\mathscr{Q})$, which corresponds to the tree shown in Figure 10b.

The second operation is a *flattening* operation, which we indicate with $\lfloor\cdot\rfloor$. Intuitively, if we have a lifted object whose leaves are themselves trees, the flattening operation "unwraps" the trees in the leaves so that they become sub-trees of said lifted object. Given a family of



(a) Before flattening. **(b)** After flattening.

Figure 11 An example of flattening.

trees $\{\mathfrak{r}_a\}_{a\in\mathscr{P}_\mathfrak{t}}$ the difference between $\mathfrak{t}[\mathfrak{r}_a]_a$ and $\lfloor\mathfrak{t}[\mathfrak{r}_a]_a\rfloor$ is therefore that the former is an element of $\mathcal{K}_\mathfrak{t}(\mathscr{T})$, whereas the latter is a proper element of $\mathscr{T}$. This operation is well-defined when $\mathsf{var}_a(\mathfrak{t})\cap\mathsf{var}(\mathfrak{r}_a)=\emptyset$ for every $a\in\mathscr{P}_\mathfrak{t}$. We can also flatten when we have have lifted objects on the right side of a composition: if we have $\mathfrak{t}[\xi_a]_a$, where $\xi_a\in\mathcal{K}_{\mathfrak{r}_a}(X)$ for every $a\in\mathscr{P}_\mathfrak{t}$, then $\lfloor\mathfrak{t}[\xi_a]_a\rfloor\in\mathcal{K}_{\lfloor\mathfrak{t}[\mathfrak{r}_a]_a\rfloor}(X)$.

A formal definition of flattening requires us to consider a slightly more general operation $\lfloor\cdot\rfloor^\mathcal{V}$, where $\mathcal{V}$ is a finite set of lifted variables which are accumulated as a lifted object is traversed. At the leaf level, $\lfloor\{x\}\rfloor^\mathcal{V}=x$ only if x is a lifted object in which none of the lifted variables in $\mathcal{V}$ occur (a condition expressed in the first line of Equation 4). In case of name clashes, the whole operation is undefined and a renaming of lifted variables is required prior to composition and flattening.

▶ **Definition 6** (Flattening). *Given $\xi\in\mathcal{K}_\mathfrak{t}(X)$, we define the* flattening of ξ under $\mathcal{V}$, *written* $\lfloor\xi\rfloor^\mathcal{V}$, *as*

$$\lfloor\{y\}\rfloor^\mathcal{V}=\begin{cases}y & \text{if } \exists\mathfrak{r},Y \text{ s.t. } y\in\mathcal{K}_\mathfrak{r}(Y)\text{ and }\mathcal{V}\cap\mathsf{var}(\mathfrak{r})=\emptyset,\\ \{y\} & \text{if } \nexists\mathfrak{r},Y \text{ s.t. } y\in\mathcal{K}_\mathfrak{r}(Y),\end{cases}\tag{4}$$
$$\lfloor u\,\{\xi_0\}\{\xi_1\}\rfloor^\mathcal{V}=u\,\{\lfloor\xi_0\rfloor^{\mathcal{V}\cup\{u\}}\}\{\lfloor\xi_1\rfloor^{\mathcal{V}\cup\{u\}}\}.$$

The actual flattening operation that we employ in the rest of the paper can then be defined as $\lfloor\cdot\rfloor=\lfloor\cdot\rfloor^\emptyset$.

▶ **Example 7.** Reconsider $\mathfrak{t}$ and $u\,\{s\,\{c_0:\mathsf{Bit}\}\{c_1:\mathsf{Bit}\}\}\{c_2:\mathsf{Bit}\}=\Delta'$ from Example 5. Let $\xi=\Delta'[s\,\{c_2:\mathsf{Bit}\}\{c_3:\mathsf{Bit}\}]^{\{u=1\}}=u\,\{s\,\{c_0:\mathsf{Bit}\}\{c_1:\mathsf{Bit}\}\}\{\{s\,\{c_2:\mathsf{Bit}\}\{c_3:\mathsf{Bit}\}\}\}\in\mathcal{K}_\mathfrak{t}(\mathscr{Q}\cup\mathcal{K}_{s\,\{\epsilon\}\{\epsilon\}}(\mathscr{Q}))$. Note that this object, corresponding to Figure 11a, is *not* properly a lifted label context, as one of its leaves is itself a lifted label context. Because s does not occur in $\mathsf{var}_{u=1}(\mathfrak{t})=\{u\}$, we can write $\lfloor\xi\rfloor=u\,\{s\,\{c_0:\mathsf{Bit}\}\{c_1:\mathsf{Bit}\}\}\{s\,\{c_2:\mathsf{Bit}\}\{c_3:\mathsf{Bit}\}\}\in\mathcal{K}_{\mathfrak{t}'}(\mathscr{Q})$, for $\mathfrak{t}'=u\,\{s\,\{\epsilon\}\{\epsilon\}\}\{s\,\{\epsilon\}\{\epsilon\}\}$. *This* is an actual lifted label context, which corresponds to Figure 11b.

In conjunction, these two operations greatly simplify our treatment of dynamic lifting, as they allow us to describe in detail the desired shape of a lifted object. For example, in later sections we often write $\Delta=\lfloor\Delta'[\Delta'',\Lambda]^{\{a\}}\rfloor$ to say that the lifted label context Δ is such that if we start from its root and follow the path described by a, we find a sub-tree Δ'',Λ (the disjoint union is lifted in a natural way, as the disjoint union of the corresponding leaves of Δ'' and Λ when these have the same underlying lifted tree), and that we are interested in only *some* elements of the corresponding lifted label context, for instance those that occur in Λ.

$$\text{id}\,\frac{}{\mathsf{input}(Q) \vdash^{\epsilon} Q \triangleright \{Q\}} \qquad \text{lift}\,\frac{C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} \ell : \mathsf{Bit} \quad a \in \mathscr{A}_{\mathfrak{t}} \quad u \notin \mathsf{var}_a(\mathfrak{t})}{C; a\,?\,\mathsf{lift}(\ell) \Rightarrow u \vdash^{\mathfrak{t} \prec^a u\,\{\epsilon\}\{\epsilon\}} Q \triangleright \Delta \prec^a u\,\{\epsilon\}\{\epsilon\}}$$

$$\text{gate}\,\frac{C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} Q' \quad a \in \mathscr{A}_{\mathfrak{t}} \quad g \in \mathscr{G}(T,U) \quad Q' \vdash_m \vec{\ell} : T \quad L \vdash_m \vec{k} : U \quad \mathsf{fresh}(\vec{k}, C)}{C; a\,?\,g(\vec{\ell}) \to \vec{k} \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} L}$$

Figure 12 The rules for CRL circuit signatures.

A Formal System for Circuit Signatures

Now that we have introduced lifting trees and lifted objects as a means to reason about dynamic lifting, we are ready to introduce the notion of *signature* of a circuit.

▶ **Definition 8** (Circuit Signature). *Given a circuit C, a lifting tree $\mathfrak{t}$, a label context Q and a lifted label context Δ, a* circuit signature *is an expression of the form $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$.*

Informally, $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$ means that C takes as input the labels in Q, performs lifting according to tree $\mathfrak{t}$ and outputs any of the leaves in Δ, which is a lifted label context backed by $\mathfrak{t}$. More formally, a valid circuit signature is derived by the rules in Figure 12. Note that $\Delta \overset{a}{;} Q'$ represents the extension of Δ with Q' on all leaves reachable by an assignment consistent with a. Formally, $\Delta \overset{a}{;} Q'$ is shorthand for $\Delta'[Q', Q_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$ if $\Delta = \Delta'[Q_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$, where $\mathscr{P}_{\mathfrak{t}}^a$ contains the paths in $\mathscr{P}_{\mathfrak{t}}$ which extend $a \in \mathscr{A}_{\mathfrak{t}}$. On the other hand, for any $\xi \in \mathcal{K}_{\mathfrak{t}}(X)$ and $\mathfrak{r} \in \mathscr{T}$ such that $\mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset$, we write $\xi \prec^a \mathfrak{r}$ to denote ξ in which every leaf x reachable by an assignment consistent with a is expanded to a sub-tree $\mathfrak{r}$ whose leaves are all x. More formally, $\xi \prec^a \mathfrak{r}$ is shorthand for $\lfloor \xi'[\mathfrak{r}[x_b]_c]_b^{\mathscr{P}_{\mathfrak{t}}^a} \rfloor$, if $\xi = \xi'[x_b]_b^{\mathscr{P}_{\mathfrak{t}}^a}$.

4 Proto-Quipper-K

We are finally ready to introduce Proto-Quipper-K, a programming language designed exactly to manipulate the kind of circuits that we presented in the previous section and guarantee the degree of flexibility that we mentioned in Section 2.

4.1 Types and Terms

The types and syntax of Proto-Quipper-K are given in Figure 13, where x and y range over the set of variable names, $u_1, \ldots, u_n$ over the set $\mathscr{V}$ of lifted variable names, $\mathfrak{t}$ over the set $\mathscr{T}$ of lifting trees and Greek letters generally indicate lifted objects. More precisely, $\alpha, \beta \in \mathcal{K}_{\mathfrak{t}}(\mathit{TYPE}), \upsilon \in \mathcal{K}_{\mathfrak{t}}(\mathit{MTYPE}), \mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERM})$ and $\lambda \in \mathcal{K}_{\mathfrak{t}}(\mathit{MVAL})$, each for some $\mathfrak{t}$. Note that parameter types are the types given to non-linear resources, i.e. duplicable values. Note also that the M-values and M-types that we introduced in Section 3 are now a proper subset of Proto-Quipper-K's values and types, respectively.

A value of the form $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ is what we called a *boxed circuit* in Section 2, that is, a datum representation of a circuit C that takes as input the labels in the tuple $\vec{\ell}$, performs lifting according to $\mathfrak{t}$ and outputs one of the possible label tuples of λ depending on the lifted variables in $\mathfrak{t}$. Correspondingly, parameter types of the form $\mathsf{Circ}_{\mathfrak{t}}(T, \upsilon)$ are called *circuit types* and represent boxed circuits. Both boxed circuits and circuit types abstract over the lifted variables in $\mathsf{var}(\mathfrak{t})$ and thus enjoy a notion of α-equivalence. The box and apply constructs are those described in Section 2 and they respectively introduce and consume boxed circuits. Specifically, the programmer is never expected to write values of the form $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ by hand.

Types	$TYPE$	A, B	$::= \mathbb{1} \mid w \mid A \multimap_{\mathfrak{t}} \beta \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \upsilon) \mid A \otimes B.$
Parameter Types	$PTYPE$	P, R	$::= \mathbb{1} \mid !\alpha \mid \mathsf{Circ}_{\mathfrak{t}}(T, \upsilon) \mid P \otimes R.$
M-types	$MTYPE$	T, U	$::= \mathbb{1} \mid w \mid T \otimes U.$
Terms	$TERM$	M, N	$::= VW \mid \mathsf{let}\ x = M\ \mathsf{in}\ \mu \mid \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M$
			$\mid \mathsf{force}\, V \mid \mathsf{box}_T\, V \mid \mathsf{apply}_{u_1,\dots,u_n}(V, W) \mid \mathsf{return}\, V.$
Values	VAL	V, W	$::= * \mid x \mid \ell \mid \lambda x_A.M \mid \mathsf{lift}\, M \mid (\vec{\ell}, C, \lambda)_{\mathfrak{t}} \mid (V, W).$
M-values	$MVAL$	$\vec{\ell}, \vec{k}$	$::= * \mid \ell \mid (\vec{\ell}, \vec{k}).$

Figure 13 Types and terms of Proto-Quipper-K.

Rather, they are expected to introduce the desired circuit by supplying an appropriate circuit-building term M to the box operator, obtaining a term of the form $\mathsf{box}_T(\mathsf{lift}\, M)$. The use of lift guarantees that M does not make use of any linear resources from the current environment, i.e. that it can be safely evaluated in an isolated environment to produce C. After a boxed circuit $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ is introduced, the programmer can potentially copy it and apply it to the underlying circuit D via a term of the form $\mathsf{apply}_{u_1,\dots,u_n}((\vec{\ell}, C, \lambda)_{\mathfrak{t}}, \vec{k})$. Such a term "unboxes" C, finds the wires identified by $\vec{k}$ among the outputs of D and appends C to them. In this process, any lifted variables in $\mathsf{var}(\mathfrak{t})$, which were abstracted in $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$, have to be instantiated with concrete names. To this end, the programmer supplies the $n = |\mathsf{var}(\mathfrak{t})|$ lifted variables $u_1, \dots, u_n$, which are expected to be fresh.

Notice that there exists a strong distinction between values and terms, the latter representing effectful computations that can introduce new lifted variables as a consequence of dynamic lifting. This choice does not detract from the expressiveness of the language, since first and foremost a value V can always be turned into an effectless computation $\mathsf{return}\, V$. Furthermore, we have that terms such as MN can still be recovered in Proto-Quipper-K as $\mathsf{let}\ x = M\ \mathsf{in}\ \{\mathsf{let}\ y = N\ \mathsf{in}\ \{xy\}\}$. The let construct is in fact a central construct in Proto-Quipper-K: on top of serving as a sequencing operator it also doubles as a conditional statement. When we evaluate the term $\mathsf{let}\ x = M\ \mathsf{in}\ \mu$, we first carry out the computation described by M, which performs dynamic lifting according to some lifting tree $\mathfrak{t}$ and consequently results in a lifted value $\phi \in \mathcal{K}_{\mathfrak{t}}(VAL)$. At this point, we are not limited to passing each and every possible value of ϕ to the *same* continuation. Rather, we can define a different continuation for every possible outcome of the liftings in $\mathfrak{t}$. To this effect, the programmer supplies a lifted term $\mu \in \mathcal{K}_{\mathfrak{t}}(TERM)$, which matches ϕ's lifting tree and thus effectively provides such a roster of continuations. The following example is meant to help convey the role of let as a control flow operator.

▶ **Example 9.** Imagine we wanted to measure qubit ℓ, dynamically lift its value into a variable u and then apply the Hadamard gate to a second qubit k *only if* $u = 1$. Suppose we had CRL definitions $ML = \mathsf{input}(\ell : \mathsf{Qubit}); \mathsf{Meas}(\ell) \to \ell'; \mathsf{lift}(\ell') \Rightarrow u$ and $H = \mathsf{input}(\ell : \mathsf{Qubit}); \mathsf{H}(\ell) \to \ell'$, corresponding to the circuit that measures and then lifts a qubit and the circuit that just applies the Hadamard gate to its input, respectively. We would write the following Proto-Quipper-K program:

$$\begin{aligned}&\mathsf{let}\ \_ = \mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, \ell)\ \mathsf{in}\\ &\quad u\,\{\mathsf{return}\, k\}\{\mathsf{apply}((\ell, H, \{\ell'\})_{\epsilon}, k)\},\end{aligned} \tag{5}$$

which may appear more familiar under the following `Haskell`-like syntactic sugar:

$$\begin{aligned}&\mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, \ell)\\ &\mathsf{when}\ (u = 1)\ \mathsf{apply}((\ell, H, \{\ell'\})_{\epsilon}, k).\end{aligned} \tag{6}$$

⌟

$$\begin{aligned}
&\lambda q_{\mathsf{Qubit}}.\,\mathsf{return}\,\lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}((\ell, H, \{\ell'\}), q)\ \mathsf{in}\ \{\\
&\qquad\qquad\qquad\qquad \mathsf{let}\ \_ = \mathsf{apply}_u((\ell, ML, u\,\{*\}\{*\})_{u\,\{\epsilon\}\{\epsilon\}}, q)\ \mathsf{in}\\
&\qquad\qquad\qquad\qquad u\,\{\mathsf{return}\,a\}\{\mathsf{apply}((\ell, Meas_\alpha, \{\ell'\}), a)\}\}
\end{aligned}$$

Figure 14 A Proto-Quipper-K program describing the circuit shown in Figure 4.

In light of this example, we can see how the circuit shown in Figure 4 can be described in Proto-Quipper-K through a program such as the one in Figure 14. To conclude this section, recall that we mentioned earlier that whenever we apply a boxed circuit we need to instantiate its lifted variables with concrete names. This process is formalized through the lifted variable analog of substitution, which we call *renaming*.

▶ **Definition 10** (Renaming of Lifted Variables). *Given a lifted variable-bearing object x and a permutation π of $\mathscr{V}$, we call π a* renaming of lifted variables *and we define $x\langle\pi\rangle$ as x in which every occurrence of a lifted variable u is replaced by $\pi(u)$. We denote by $s_1/u_1, \ldots, s_n/u_n$ a permutation that exchanges s_1 for $u_1, \ldots s_n$ for u_n.*

4.2 Proto-Quipper-K's Typing Rules

At its core, Proto-Quipper-K's type system is a linear type system which distinguishes between *computations* (i.e. terms), which can be effectful, and *values.* We therefore introduce two kinds of typing judgments. One for terms, which are given lifted types, and one for values, which have regular types.

▶ **Definition 11** (Typing Judgments). *Given a typing context Γ, a label context Q, a term M and a lifted type α, a* computational typing judgment *is an expression of the form*

$$\Gamma; Q \vdash_c^{\mathfrak{t}} M : \alpha. \tag{7}$$

Given Γ, Q, a value V and a type A, a value typing judgment *is an expression of the form*

$$\Gamma; Q \vdash_v V : A. \tag{8}$$

When a typing context contains *exclusively* variables with parameter types, we write it as Φ, but in general a typing context Γ can contain both linear and parameter variables. Typing judgments are derived via the rules in Figure 15, where $s_1, \ldots, s_n$ range over $\mathscr{V}$. Note that we assume that $\mathscr{V}$ is totally ordered, so that when we write $\mathsf{var}(\mathfrak{t}) = \{u_1, \ldots, u_n\}$, e.g. in the *apply* rule, we have $u_1 \leq u_2 \leq \cdots \leq u_n$. The relational symbols $\Vdash_c$ and $\Vdash_v$ denote the lifting of the computation and value typing judgments to some tree $\mathfrak{t}$.

▶ **Definition 12** (Lifted Typing Judgments). *Given a lifting tree $\mathfrak{t}$, if for all $a \in \mathscr{P}_{\mathfrak{t}}$ we have $\Gamma_a; Q_a \vdash_c^{\mathfrak{r}_a} M_a : \alpha_a$, then we write*

$$\mathfrak{t}[\Gamma_a]_a; \mathfrak{t}[Q_a]_a \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mathfrak{t}[M_a]_a : \mathfrak{t}[\alpha_a]_a. \tag{9}$$

If for all $a \in \mathscr{P}_{\mathfrak{t}}$ we have $\Gamma_a; Q_a \vdash_v V_a : A_a$, then we write

$$\mathfrak{t}[\Gamma_a]_a; \mathfrak{t}[Q_a]_a \Vdash_v^{\mathfrak{t}} \mathfrak{t}[V_a]_a : \mathfrak{t}[A_a]_a. \tag{10}$$

For convenience, within such judgments, every lifted object (e.g. Δ in $\emptyset; \Delta \Vdash_v^{\mathfrak{t}} \lambda : \upsilon$ in the *circ* rule) is assumed to be backed by $\mathfrak{t}$, whereas every component which is *not* a lifted object (e.g. Φ or x in $\Phi, \Gamma_2, x : \alpha; Q_2 \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mu : \theta$ in the *let* rule) is assumed to be constant across the branches of $\mathfrak{t}$.

$$\text{unit}\,\frac{}{\Phi;\emptyset \vdash_v * : \mathbb{1}} \qquad \text{var}\,\frac{}{\Phi, x : A;\emptyset \vdash_v x : A} \qquad \text{label}\,\frac{}{\Phi;\ell : w \vdash_v \ell : w}$$

$$\text{abs}\,\frac{\Gamma, x : A; Q \vdash_c^{\mathfrak{t}} M : \beta}{\Gamma; Q \vdash_v \lambda x_A.M : A \multimap_{\mathfrak{t}} \beta} \qquad \text{app}\,\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \multimap_{\mathfrak{t}} \beta \quad \Phi, \Gamma_2; Q_2 \vdash_v W : A}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathfrak{t}} VW : \beta}$$

$$\text{let}\,\frac{\Phi, \Gamma_1; Q_1 \vdash_c^{\mathfrak{t}} M : \alpha \quad \mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERM}) \quad \Phi, \Gamma_2, x : \alpha; Q_2 \Vdash_c^{\mathfrak{t}[\mathfrak{r}_a]_a} \mu : \theta}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\lfloor \mathfrak{t}[\mathfrak{r}_a]_a \rfloor} \mathsf{let}\ x = M\ \mathsf{in}\ \mu : \lfloor \theta \rfloor}$$

$$\text{tuple}\,\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \quad \Phi, \Gamma_2; Q_2 \vdash_v W : B}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_v (V, W) : A \otimes B}$$

$$\text{dest}\,\frac{\Phi, \Gamma_1; Q_1 \vdash_v V : A \otimes B \quad \Phi, \Gamma_2, x : A, y : B; Q_2 \vdash_c^{\mathfrak{t}} M : \alpha}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathfrak{t}} \mathsf{let}\ (x, y) = V\ \mathsf{in}\ M : \alpha} \qquad \text{lift}\,\frac{\Phi;\emptyset \vdash_c^{\epsilon} M : \alpha}{\Phi;\emptyset \vdash_v \mathsf{lift}\, M : !\alpha}$$

$$\text{force}\,\frac{\Gamma; Q \vdash_v V : !\alpha}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{force}\, V : \alpha} \qquad \text{box}\,\frac{\Gamma; Q \vdash_v V : !\{T \multimap_{\mathfrak{t}} \upsilon\}}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{box}_T\, V : \{\mathsf{Circ}_{\mathfrak{t}}(T, \upsilon)\}}$$

$$\text{apply}\,\frac{\begin{array}{c}\Phi, \Gamma_1; Q_1 \vdash_v V : \mathsf{Circ}_{\mathfrak{t}}(T, \upsilon) \quad \Phi, \Gamma_2; Q_2 \vdash_v W : T \\ \mathsf{var}(\mathfrak{t}) = \{u_1, \dots, u_n\} \qquad \pi = s_1/u_1, \dots, s_n/u_n\end{array}}{\Phi, \Gamma_1, \Gamma_2; Q_1, Q_2 \vdash_c^{\mathfrak{t}\langle\pi\rangle} \mathsf{apply}_{s_1,\dots,s_n}(V, W) : \upsilon\langle\pi\rangle}$$

$$\text{circ}\,\frac{C \vdash^{\mathfrak{t}} Q \triangleright \Delta \quad \emptyset; Q \vdash_v \vec{\ell} : T \quad \emptyset; \Delta \Vdash_v^{\mathfrak{t}} \lambda : \upsilon}{\Phi;\emptyset \vdash_v (\vec{\ell}, C, \lambda)_{\mathfrak{t}} : \mathsf{Circ}_{\mathfrak{t}}(T, \upsilon)} \qquad \text{return}\,\frac{\Gamma; Q \vdash_v V : A}{\Gamma; Q \vdash_c^{\epsilon} \mathsf{return}\, V : \{A\}}$$

Figure 15 The typing rules of Proto-Quipper-K.

The *let* rule is unsurprisingly the most interesting rule of the system: if M is a term of lifted type $\alpha \in \mathcal{K}_{\mathfrak{t}}(\mathit{TYPE})$, μ is a lifted term with the same underlying tree structure $\mathfrak{t}$ and for every root-to-leaf path a in $\mathfrak{t}$ we have $\Phi, \Gamma_2, x : \alpha; Q_2 \vdash_c^{\mathfrak{r}_a} \mu(a) : \theta(a)$ (note that this last condition is captured by the lifted typing judgment), then the lifted type of $\mathsf{let}\ x = M\ \mathsf{in}\ \mu$ is obtained by simply flattening the generic lifted object θ to $\lfloor \theta \rfloor \in \mathcal{K}_{\lfloor \mathfrak{t}[\mathfrak{r}_a]_a \rfloor}(\mathit{TYPE})$. The *apply* rule also plays a pivotal role, as it actually introduces lifting into the type of a term. If V is a value of type $\mathsf{Circ}_{\mathfrak{t}}(T, \upsilon)$, and thus corresponds to a boxed circuit which performs lifting according to $\mathfrak{t}$, then applying V to some appropriate wires W of the underlying circuit introduces the same lifting pattern into the computation, which has type $\upsilon\langle\pi\rangle \in \mathcal{K}_{\mathfrak{t}\langle\pi\rangle}(\mathit{MTYPE})$. The renaming of lifted variables π is required to avoid name clashes.

An example of a type derivation that leverages the full expressiveness of Proto-Quipper-K's type system is the one for the program shown in Figure 14 (describing the circuit in Figure 4), which can be found in Appendix A. To conclude this section, notice how just like M-values and M-types are a subset of the values and types of Proto-Quipper-K, the type system for M-values is in a one-to-one correspondence with a subset of Proto-Quipper-K's type system.

4.3 A Big-Step Operational Semantics

The big-step operational semantics of the language is based on an evaluation relation $(C, a, M) \Downarrow (D, \phi)$, where ϕ is a lifted value. This means that the term M evaluates to one of the possible values in ϕ, depending on the outcome of intermediate measurements, and updates the underlying circuit C upon branch a as a side effect, obtaining an updated circuit D. We call (C, a, M) a *left configuration* and (D, ϕ) a *right configuration*. Before we give the actual rules of the semantics, we must first give some definitions.

First and foremost, note that we are not as interested in the actual names of labels within a boxed circuit as much as we are in the structure that they convey. In fact, when we apply circuits to one another, it might be necessary to rename some of the labels occurring in the applicand in order to avoid naming conflicts, all the while preserving its structure. For this reason, whenever two circuits share the same structure and only differ by their respective labels, we consider them to be equivalent.

▶ **Definition 13** (Equivalent Boxed Circuits). *We say that two boxed circuits* $(\vec{\ell}, C, \lambda)_{\mathfrak{t}}$ *and* $(\vec{\ell'}, C', \lambda')_{\mathfrak{t}}$ *are* equivalent*, and we write* $(\vec{\ell}, C, \lambda)_{\mathfrak{t}} \cong (\vec{\ell'}, C', \lambda')_{\mathfrak{t}}$*, when they only differ by a renaming of labels.*

Next, we define the two operations that actually implement the semantics of apply. The circuit insertion function $::_a$ is just a simple concatenation function defined on CRL circuits and all the heavy lifting is actually performed by the append function. This function is responsible for the actual unboxing of a boxed circuit, the renaming of its labels (to match the outputs of the underlying circuit and avoid labeling conflicts), the instantiation of the abstracted lifted variable names within it and its insertion in the underlying circuit.

▶ **Definition 14** (Insertion of Circuits). *Suppose C and D are two circuits. We define* the insertion of D in C on branch a*, written $C ::_a D$ as:*

$$\begin{aligned} C ::_a \mathsf{input}(Q) &= C, \\ C ::_a (D'; b \mathbin{?} g(\vec{\ell}) \to \vec{k}) &= (C ::_a D'); a \cup b \mathbin{?} g(\vec{\ell}) \to \vec{k}, \\ C ::_a (D'; b \mathbin{?} \mathsf{lift}(\ell) \Rightarrow u) &= (C ::_a D'); a \cup b \mathbin{?} \mathsf{lift}(\ell) \Rightarrow u, \end{aligned} \tag{11}$$

where $a \cup b$ denotes the union of two assignments with disjoint domains.

▶ **Definition 15** (append). *Suppose $C \vdash^{\mathfrak{t}} Q \triangleright \Delta$ is a circuit and $(\vec{\ell}, D, \lambda)_{\mathfrak{r}}$ is a boxed circuit with* $\mathsf{var}(\mathfrak{r}) = \{u_1, \ldots, u_n\}$*. Suppose $a \in \mathscr{P}_{\mathfrak{t}}$ and let $\vec{k}$ be a label tuple whose labels all occur in $\Delta(a)$. Finally, let $s_1, \ldots, s_n$ be a sequence of distinct lifted variable names which do not occur in* $\mathsf{var}_a(\mathfrak{t})$*. We define* $\mathsf{append}(C, a, \vec{k}, (\vec{\ell}, D, \lambda)_{\mathfrak{r}}, s_1, \ldots, s_n)$ *as the function that*

1. *Finds $(\vec{k}, D', \lambda')_{\mathfrak{t}} \cong (\vec{\ell}, D, \lambda)_{\mathfrak{t}}$ such that all the labels occurring in D', but not in $\vec{k}$, are fresh in C,*
2. *Computes $D'' = D'\langle s_1/u_1, \ldots, s_n/u_n\rangle$ and $\lambda'' = \lambda'\langle s_1/u_1, \ldots, s_n/u_n\rangle$,*
3. *Returns $(C ::_a D'', \lambda'')$.*

The rules of the operational semantics can be found in Figure 16, where $\mathsf{freshlabels}(T)$ produces a pair $(Q, \vec{\ell})$ such that $Q \vdash_m \vec{\ell} : T$. Note that, for each $\mathfrak{t}$, we assume to have a total order over the elements of $\mathscr{P}_{\mathfrak{t}}$, so that when we write $\mathscr{P}_{\mathfrak{t}} = \{a_1, \ldots, a_n\}$ we have $a_1 \leq a_2 \leq \cdots \leq a_n$ and the semantics is deterministic. In order to prove progress in Section 5, we also consider a notion of divergence for configurations. Intuitively, a configuration (C, a, M) *diverges*, and we write $(C, a, M) \Uparrow$, when its evaluation does not terminate. More formally, divergence is defined coinductively by means of the rules in Figure 17.

5 Type Soundness

In general, the well-typedness of a Proto-Quipper-K configuration strongly depends on the underlying circuit. Specifically, a term M is well-typed when all the free labels occurring in it can be found with the appropriate type in the outputs of the underlying circuit, on the branch that M is manipulating. For this reason, we give the following notions of well-typedness.

$$\text{app}\,\dfrac{(C, a, M[V/x]) \Downarrow (D, \phi)}{(C, a, (\lambda x_A.M)V) \Downarrow (D, \phi)} \qquad \text{dest}\,\dfrac{(C, a, M[V/x, W/y]) \Downarrow (D, \phi)}{(C, a, \mathsf{let}\ (x, y) = (V, W)\ \mathsf{in}\ M) \Downarrow (D, \phi)}$$

$$\text{let}\,\dfrac{\begin{array}{c}(C, a, M) \Downarrow (C_1, \phi) \qquad \phi \in \mathcal{K}_{\mathfrak{t}}(\mathit{VAL}) \qquad \mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERM}) \\ \mathscr{P}_{\mathfrak{t}} = \{a_1, \dots, a_n\} \quad (C_i, a \cup a_i, \mu(a_i)[\phi(a_i)/x]) \Downarrow (C_{i+1}, \psi_{a_i})\ \text{ for }\ i = 1, \dots, n\end{array}}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Downarrow (C_{n+1}, \lfloor \mathfrak{t}[\psi_a]_a \rfloor)}$$

$$\text{force}\,\dfrac{(C, a, M) \Downarrow (D, \phi)}{(C, a, \mathsf{force}(\mathsf{lift}\, M)) \Downarrow (D, \phi)} \qquad \text{apply}\,\dfrac{(C', \lambda') = \mathsf{append}(C, a, \vec{k}, (\vec{\ell}, D, \lambda)_{\mathfrak{t}}, s_1, \dots, s_n)}{(C, a, \mathsf{apply}_{s_1, \dots, s_n}((\vec{\ell}, D, \lambda)_{\mathfrak{t}}, \vec{k})) \Downarrow (C', \lambda')}$$

$$\text{box}\,\dfrac{(Q, \vec{\ell}) = \mathsf{freshlabels}(T) \quad (\mathsf{input}(Q), \emptyset, \mathsf{let}\ x = M\ \mathsf{in}\ \{x\vec{\ell}\}) \Downarrow (D, \lambda) \quad \lambda \in \mathcal{K}_{\mathfrak{t}}(\mathit{MVAL})}{(C, a, \mathsf{box}_T(\mathsf{lift}\, M)) \Downarrow (C, \{(\vec{\ell}, D, \lambda)_{\mathfrak{t}}\})}$$

$$\text{return}\,\dfrac{}{(C, a, \mathsf{return}\, V) \Downarrow (C, \{V\})}$$

Figure 16 The big-step operational semantics of Proto-Quipper-K.

$$\text{app}\,\dfrac{(C, a, M[V/x]) \Uparrow}{(C, a, (\lambda x_A.M)V) \Uparrow} \qquad \text{dest}\,\dfrac{(C, a, M[V/x, W/y]) \Uparrow}{(C, a, \mathsf{let}\ (x, y) = (V, W)\ \mathsf{in}\ M) \Uparrow}$$

$$\text{force}\,\dfrac{(C, a, M) \Uparrow}{(C, a, \mathsf{force}(\mathsf{lift}\, M)) \Uparrow} \qquad \text{let-now}\,\dfrac{(C, a, M) \Uparrow}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Uparrow}$$

$$\text{let-then}\,\dfrac{\begin{array}{c}(C, a, M) \Downarrow (C_1, \phi) \qquad \phi \in \mathcal{K}_{\mathfrak{t}}(\mathit{VAL}) \qquad \mu \in \mathcal{K}_{\mathfrak{t}}(\mathit{TERM}) \\ \mathscr{P}_{\mathfrak{t}} = \{a_1, \dots, a_n\} \quad (C_i, a \cup a_i, \mu(a_i)[\phi(a_i)/x]) \Downarrow (C_{i+1}, \psi_{a_i})\ \text{ for }\ i = 1, \dots, j-1 \\ (C_j, a \cup a_j, \mu(a_j)[\phi(a_j)/x]) \Uparrow\end{array}}{(C, a, \mathsf{let}\ x = M\ \mathsf{in}\ \mu) \Uparrow}$$

$$\text{box}\,\dfrac{(Q, \vec{\ell}) = \mathsf{freshlabels}(T) \quad (\mathsf{input}(Q), \emptyset, \mathsf{let}\ x = M\ \mathsf{in}\ \{x\vec{\ell}\}) \Uparrow}{(C, a, \mathsf{box}_T(\mathsf{lift}\, M)) \Uparrow}$$

Figure 17 The big-step divergence rules of Proto-Quipper-K.

▶ **Definition 16** (Well-Typed Configuration). *We say that*

- *a left configuration* (C, a, M) *is* well-typed with input Q, past lifting tree $\mathfrak{t}$, future lifting tree $\mathfrak{r}$, lifted type α and outputs Δ, *and we write* $Q \vdash^{\mathfrak{r}}_{\mathfrak{t}} (C, a, M) : \alpha; \Delta$, *when* $a \in \mathscr{P}_{\mathfrak{t}}, \mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset, C \vdash^{\mathfrak{t}} Q \triangleright \Delta \overset{a}{;} Q'$ *and* $\emptyset; Q' \vdash^{\mathfrak{r}}_{c} M : \alpha$*,*
- *a right configuration* (C, ϕ) *is* well-typed in the a branch with input Q, overall lifting tree $\mathfrak{t}$, lifted type α and outputs Δ*, and we write* $Q \vdash^{a}_{\mathfrak{t}} (C, \phi) : \alpha; \Delta$*, when* $\mathfrak{t} = \lfloor \mathfrak{t}'[\mathfrak{r}]^{\{a\}} \rfloor, \Delta = \lfloor \Delta'[\Delta'']^{\{a\}} \rfloor, C \vdash^{\mathfrak{t}} Q \triangleright \lfloor \Delta'[\Delta'', \Lambda]^{\{a\}} \rfloor$ *and* $\emptyset; \Lambda \Vdash^{\mathfrak{r}}_{v} \phi : \alpha$*.*

That being said, we are mainly interested in *closed* computations, in which the evaluation of a term builds the underlying circuit entirely from scratch. That is, we are interested in computations that start from configurations of the form $(\mathsf{input}(\emptyset), \emptyset, M)$, for some M. Whenever $(\mathsf{input}(\emptyset), \emptyset, M) \Downarrow (C, \phi)$ for some C, ϕ, we simply write $M \Downarrow (C, \phi)$ and whenever $(\mathsf{input}(\emptyset), \emptyset, M) \Uparrow$ we write $M \Uparrow$. In the same guise, we say that M is a *well-typed term with lifted type* α *depending on* $\mathfrak{t}$, and we write $\vdash^{\mathfrak{t}} M : \alpha$, whenever $\emptyset \vdash^{\mathfrak{t}}_{\epsilon} (\mathsf{input}(\emptyset), \emptyset, M) : \alpha; \{\emptyset\}$, while we say that (C, ϕ) is a *well-typed closed configuration with lifted type* α *depending on* $\mathfrak{t}$, and we write $\vdash^{\mathfrak{t}} (C, \phi) : \alpha$, whenever $\emptyset \vdash^{\emptyset}_{\mathfrak{t}} (C, \phi) : \alpha; \mathfrak{t}[\emptyset]_b$. We are now ready to give the relevant type safety results for Proto-Quipper-K.

▶ **Theorem 17** (Subject Reduction). *If* $\vdash^{\mathfrak{t}} M : \alpha$ *and* $\exists C, \phi$ s.t. $M \Downarrow (C, \phi)$*, then* $\vdash^{\mathfrak{t}} (C, \phi) : \alpha$.

Proof sketch. We prove the more general claim that whenever $Q \vdash^{\mathfrak{t}}_{\mathfrak{r}} (D, a, M) : \alpha; \Delta$ and $\exists C, \phi$ s.t. $(D, a, M) \Downarrow (C, \phi)$, then $Q \vdash^{a}_{\lfloor \mathfrak{r}[\mathfrak{t}]^{\{a\}} \rfloor} (C, \phi) : \alpha; \Delta \prec^{a} \mathfrak{t}$, from which we obtain the subject reduction claim by choosing $D = \mathsf{input}(\emptyset), a = \emptyset, Q = \emptyset, \mathfrak{r} = \epsilon$ and $\Delta = \{\emptyset\}$. We proceed by induction on $(D, a, M) \Downarrow (C, \phi)$ and case analysis on the last rule used in its derivation. A number of cases require additional lemmata, more specifically:

1. A substitution lemma is naturally required for the *app, dest, let* cases. In our scenario, this amounts to proving that whenever $\Phi, \Gamma'; Q' \vdash_v V : A$ and Π is a type derivation, then
   a. If the conclusion of Π is $\Phi, \Gamma, x : A; Q \vdash^{\mathfrak{t}}_{c} M : \alpha$, then $\Phi, \Gamma, \Gamma'; Q, Q' \vdash^{\mathfrak{t}}_{c} M[V/x] : \alpha$,
   b. If the conclusion of Π is $\Phi, \Gamma, x : A; Q \vdash_v W : B$, then $\Phi, \Gamma, \Gamma'; Q, Q' \vdash_v W[V/x] : B$,

   We prove the claim separately for the cases in which V has parameter type and linear type. In both cases, we proceed by induction on the size of Π and case analysis on its last rule.
2. The *apply* case is particularly delicate and requires us to prove that all the operations performed by the append function on the applicand and the underlying circuit alter their respective circuit signatures in a predictable way. More specifically:
   a. For the first step of append, we show that equivalent circuits have the same type, that is, that whenever $\emptyset; \emptyset \vdash_v (\vec{\ell}, C, \lambda)_{\mathfrak{t}} : \mathsf{Circ}_{\mathfrak{t}}(T, U)$ and $(\vec{\ell}, C, \lambda)_{\mathfrak{t}} \cong (\vec{\ell}', C', \lambda')_{\mathfrak{t}}$, then $\emptyset; \emptyset \vdash_v (\vec{\ell}', C', \lambda')_{\mathfrak{t}} : \mathsf{Circ}_{\mathfrak{t}}(T, U)$. This reflects the idea that the type of a circuit depends on its structure, and not on the specific labels used to convey said structure.
   b. Similarly, for the second step, we show that lifted variable renaming preserves circuit signatures and lifted typing judgments, that is, that for every renaming of lifted variables π, $C \vdash^{\mathfrak{t}} Q \rhd \Delta$ implies $C\langle\pi\rangle \vdash^{\mathfrak{t}\langle\pi\rangle} Q \rhd \Delta\langle\pi\rangle$ and $\gamma; \Delta \Vdash^{\mathfrak{t}}_{v} \phi : \alpha$ implies $\gamma\langle\pi\rangle; \Delta\langle\pi\rangle \Vdash^{\mathfrak{t}\langle\pi\rangle}_{v} \phi\langle\pi\rangle : \alpha\langle\pi\rangle$, where γ is a lifted context in $\mathcal{K}_{\mathfrak{t}}(\mathit{CONTEXT})$. Similarly to the previous point, this reflects the idea that the structure of a lifted object is more important than the specific lifted variable names used to convey said structure.
   c. For the third step, we show that whenever we have an underlying circuit C and an applicand D whose labels and lifted variables have already been renamed appropriately, the concatenated circuit $C ::_a D$ is such that its output on branch a contains both the outputs of D and the outputs of C that D was *not* applied on. That is, we prove that whenever $C \vdash^{\mathfrak{t}} Q \rhd \Delta[Q', Q'']^{\{a\}}$ for some $a \in \mathscr{P}_{\mathfrak{t}}$ and $D \vdash^{\mathfrak{r}} Q' \rhd \Lambda$ for some D such that the labels that occur in D, but not in Q', are fresh in C and $\mathsf{var}_a(\mathfrak{t}) \cap \mathsf{var}(\mathfrak{r}) = \emptyset$, then $C ::_a D \vdash^{\lfloor \mathfrak{t}[\mathfrak{r}]^{\{a\}} \rfloor} Q \rhd \lfloor \Delta[\Lambda, Q'']^{\{a\}} \rfloor$. We prove this by induction on $D \vdash^{\mathfrak{r}} Q' \rhd \Lambda$. This result clearly allows us to conclude the subject reduction claim for the *apply* case.
3. The *let* case is also particularly delicate, although more technical than the *apply* case: it requires us to apply the inductive hypothesis once for the evaluation of the left side of the let and n times for the evaluation of each of the possible branches on the right side, which happens in sequence. This means that the conclusion of each application of the inductive hypothesis must become the premise of the following application. More specifically, for every well-typed right configuration (C_{i+1}, ψ_{a_i}) we must be able to prove that $(C_{i+1}, a \cup a_{i+1}, \mu(a_{i+1})[\phi(a_{i+1})/x])$ is a well-typed left configuration. A key lemma in this process tells us that for any two generic lifted objects $\xi \in \mathcal{K}_{\mathfrak{t}}(X)$ and $\theta \in \mathcal{K}_{\mathfrak{r}}(X)$ and any two assignments $a \in \mathscr{P}_{\mathfrak{t}}$ and $b \in \mathscr{A}_{\mathfrak{r}}$, we have $\lfloor \xi[\theta[x_c]_c^{\mathscr{P}^b_{\mathfrak{r}}}]^{\{a\}} \rfloor = \lfloor \xi[\theta]^{\{a\}} \rfloor [x_c]_c^{\mathscr{P}^{a \cup b}_{\lfloor \mathfrak{t}[\mathfrak{r}]^{\{a\}} \rfloor}}$. ◀

▶ **Theorem 18** (Progress). *If* $\vdash^{\mathfrak{t}} M : \alpha$*, then either* $\exists C, \phi$ s.t. $M \Downarrow (C, \phi)$ *or* $M \Uparrow$.

Proof sketch. We consider the equivalent claim that if $\vdash^{\mathfrak{t}} M : \alpha$ and $\nexists C, \phi . M \Downarrow (C, \phi)$, then $M \Uparrow$. We then prove the more general result that if $Q \vdash^{\mathfrak{t}}_{\mathfrak{r}} (D, a, M) : \alpha; \Delta$ and $\nexists C, \phi . (D, a, M) \Downarrow (C, \phi)$, then $(D, a, M) \Uparrow$, from which we obtain the progress claim by choosing $D = \mathsf{input}(\emptyset), a = \emptyset, Q = \emptyset, \mathfrak{r} = \epsilon$ and $\Delta = \{\emptyset\}$. We proceed by coinduction and case analysis on M. The proof is fairly straightforward and makes extensive use of the general subject reduction theorem and of some of its lemmata. The one interesting case is *apply*, which is proven vacuously by showing that append is always defined under the hypothesis:

1. The first step of append must find a circuit $(\vec{k}, D', \lambda')_{\mathfrak{t}} \cong (\vec{\ell}, D, \lambda)_{\mathfrak{t}}$ such that the labels in $\vec{k}$ are given and correspond to the target labels in the underlying circuit C and all the other labels in D' are fresh in C. A key lemma tells us that since $\vec{\ell}$ and $\vec{k}$ have the same M-type T, then it is possible to rename the former to the latter. It is then straightforward to extend this renaming to all the labels occurring in D in a way that fulfills the aforementioned requirements.
2. The second step must compute $D'\langle\pi\rangle$ and $\lambda'\langle\pi\rangle$. This is always possible since π is a valid renaming of lifted variables and thus a permutation.
3. The last step must compute $C ::_a D'\langle\pi\rangle$. We show that for every assignment b occurring in $D'\langle\pi\rangle$ it holds that $\mathsf{dom}(a) \cap \mathsf{dom}(b) = \emptyset$. Therefore, the concatenation operation is defined and append returns $(C ::_a D'\langle\pi\rangle, \lambda'\langle\pi\rangle)$. ◀

Detailed proofs of subject reduction (Theorem 17) and progress (Theorem 18) can be found in the extended version of the paper [2].

6 Conclusion

Our Contributions

This paper introduces a new paradigmatic language for dynamic lifting belonging to the Proto-Quipper family of languages. The language, called Proto-Quipper-K, can be seen as an extension of Proto-Quipper-M which allows for the Boolean information flowing within bit wires to be lifted from the circuit level to the program level. In order to make the circuit construction process as flexible and as general as possible, a powerful type and effect system based on lifting trees has been introduced. This allows for the typing of programs which produce highly non-uniform circuits, making Proto-Quipper-K strictly more expressive than Quipper and potentially useful in scenarios such as one-way quantum computing [4]. Although the use of the syntax introduced in this paper is not *essential* to implement the measurement patterns encountered in one-way quantum computing (which, after all, can be simulated by regular quantum circuits), the non-uniform approach to dynamic lifting that we adopt in our work would allow for a higher degree of flexibility in building and manipulating patterns. In other words, even when the circuit produced at the end of the computation is a uniform circuit, we can build it incrementally, going through non-uniform circuits. The main technical results we obtained for Proto-Quipper-K are type soundness, in the sense of subject reduction and progress theorems.

Future Work

In this paper we focused on the operational aspects, leaving an investigation about a possible denotational account of Proto-Quipper-K as future work. A related problem is that of understanding the precise nature of the lifting operation that we use pervasively in the paper. In particular, while it would be tempting to interpret $\mathcal{K}_{\mathfrak{t}}$ as a graded monad [8, 12] with unit $\eta : X \to \mathcal{K}_{\epsilon}(X)$ defined as $\eta\, x = \{x\}$ and multiplication $\lfloor\cdot\rfloor : \mathcal{K}_{\mathfrak{t}}(\mathcal{K}_{\mathfrak{r}}(X)) \to \mathcal{K}_{\mathfrak{t}[\mathfrak{r}]_a}(X)$, we generally work with objects that do not belong to $\mathcal{K}_{\mathfrak{t}}(\mathcal{K}_{\mathfrak{r}}(X))$ for some fixed $\mathfrak{t}, \mathfrak{r}$, so we find that this interpretation is not entirely appropriate, at least if we assume grades to be elements of *one fixed monoid*.

Another aspect that we left open is the consolidation of homogeneous branches in a computation. In many contexts, it is extremely natural to ask that the type of a term be uniform with respect to a certain lifted variable u, i.e. that it be of the form $u\,\{\alpha\}\{\alpha\}$. If a term M were typed this way, it would be natural to construct a new term $\nu u.M$ in which u is no longer lifted and which could thus be given the type α. We claim that such a local name binder can be added to the language without substantially altering its metatheory.

The last problem that we deliberately leave open concerns the notion of generalized circuits used in this paper. On the one hand, it can certainly be said that it can model quantum circuits in their full generality, including those measurement patterns found in measurement-based quantum computing [4]. On the other hand, while it is clear that such circuits make computational sense (after all, any such computation can be simulated by a classically controlled quantum Turing machine [18]), it is not clear what kind of correspondence exists between them and quantum circuits in their usual form [15], which is the one most often considered in the literature.

Related Work

As already mentioned in the introduction, various paradigmatic λ-calculi modeling the Quipper programming language have been introduced in the literature [6, 7, 13, 20, 21]. In this work, we took inspiration from Proto-Quipper-M [20], which however cannot handle dynamic lifting. The only members of the Proto-Quipper family that can handle dynamic lifting, in a uniform and more restricted form than ours, are Proto-Quipper-L [13] and Proto-Quipper-Dyn [6], which have been introduced very recently and independently of this work. Interestingly, the class of circuits targeted by Proto-Quipper-L– which the authors have named *quantum channels* – indeed includes non-uniform circuits like the one in Figure 4, which leads us to believe that they actually match our generalized quantum circuits in terms of expressiveness. However, Proto-Quipper-L's type system rejects the programs that build such circuits. On the other hand, Proto-Quipper-Dyn follows a different approach, with two interleaved operational semantics: one for circuit building, which only happens inside a boxing operator and does *not* support dynamic lifting, and one for circuit execution, where dynamic lifting is allowed. Therefore, as a circuit description language, Proto-Quipper-Dyn targets traditional circuits. At the type level, this distinction is reflected in a modal type system which keeps track of whether dynamic lifting is used, and therefore whether a circuit-building function can be boxed or not. That being said, most of the aforementioned contributions differ from ours in that they focus mainly on the denotational semantics of the language rather than its operational semantics.

The type and effect system paradigm is well known from the literature [3, 12, 14, 16] and has been used in various contexts as a way to reflect information on the effects produced by a program in its type. In our case, the relevant effect is a choice effect, which is mirrored both in the operational semantics and in the type system. As already stated, the problem of giving a proper monadic status to the considered choice effect remains open, although this work shows that *operationally speaking* everything works smoothly.

Finally, it is worth mentioning that the circuit description paradigm is not the only approach to designing quantum programming languages and calculi. For instance, QCL [25], QML [1] and Selinger and Valiron's quantum λ-calculus [23] are some examples of quantum programming languages whose instructions are designed to be executed immediately on quantum hardware, without any direct reference to quantum circuits.

References

1 Thorsten Altenkirch and Jonathan Grattage. A functional quantum programming language. In *Proc. of LICS*, 2005. doi:10.1109/lics.2005.1.

2 Andrea Colledan and Ugo Dal Lago. On dynamic lifting and effect typing in circuit description languages (extended version), 2022. arXiv:2202.07636.

3 Ugo Dal Lago and Charles Grellois. Probabilistic termination by monadic affine sized typing. *ACM Trans. Program. Lang. Syst.*, 41(2), March 2019. doi:10.1145/3293605.

4 Vincent Danos, Elham Kashefi, and Prakash Panangaden. The measurement calculus. *J. ACM*, 54(2), April 2007. doi:10.1145/1219092.1219096.

5 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. A tutorial introduction to quantum circuit programming in dependently typed proto-quipper. In *Proc. of RC*, Berlin, Heidelberg, 2020. Springer-Verlag. doi:10.1007/978-3-030-52482-1_9.

6 Peng Fu, Kohei Kishida, Neil J. Ross, and Peter Selinger. Proto-quipper with dynamic lifting. *Proc. ACM Program. Lang.*, 7(POPL), January 2023. doi:10.1145/3571204.

7 Peng Fu, Kohei Kishida, and Peter Selinger. Linear dependent type theory for quantum programming languages: Extended abstract. In *Proc. of LICS*, 2020. doi:10.1145/3373718.3394765.

8 Soichiro Fujii, Shin-ya Katsumata, and Paul-André Melliès. Towards a formal theory of graded monads. In *Proc. of FoSSaCS*, Berlin, Heidelberg, 2016. doi:10.1007/978-3-662-49630-5_30.

9 Simon J. Gay. Quantum programming languages: Survey and bibliography. *Math. Struct. Comput. Sci.*, 16(4):581–600, August 2006. doi:10.1017/S0960129506005378.

10 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. An introduction to quantum programming in quipper. In *Proc. of RC*, pages 110–124, 2013. doi:10.1007/978-3-642-38986-3_10.

11 Alexander S. Green, Peter LeFanu Lumsdaine, Neil J. Ross, Peter Selinger, and Benoît Valiron. Quipper. In *Proc. of PLDI*, pages 333–342, June 2013. doi:10.1145/2499370.2462177.

12 Shin-ya Katsumata. Parametric effect monads and semantics of effect systems. In *Proc. of POPL*, pages 633–645, January 2014. doi:10.1145/2578855.2535846.

13 Dongho Lee, Valentin Perrelle, Benoît Valiron, and Zhaowei Xu. Concrete Categorical Model of a Quantum Circuit Description Language with Measurement. In *Proc. of FSTTCS*, volume 213 of *LIPIcs*, pages 51:1–51:20, 2021. doi:10.4230/LIPIcs.FSTTCS.2021.51.

14 Alan Mycroft, Dominic Orchard, and Tomas Petricek. Effect systems revisited—control-flow algebra and semantics. In Christian W. Probst, Chris Hankin, and René Rydhof Hansen, editors, *Semantics, Logics, and Calculi: Essays Dedicated to Hanne Riis Nielson and Flemming Nielson on the Occasion of Their 60th Birthdays*, pages 1–32. Springer International Publishing, Cham, 2016. doi:10.1007/978-3-319-27810-0_1.

15 Michael A. Nielsen and Isaac L. Chuang. *Quantum Computation and Quantum Information: 10th Anniversary Edition.* Cambridge University Press, 2010. doi:10.1017/CBO9780511976667.

16 Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In Ernst-Rüdiger Olderog and Bernhard Steffen, editors, *Correct System Design: Recent Insights and Advances*, pages 114–136. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-48092-7_6.

17 Jens Palsberg. Toward a universal quantum programming language. *XRDS: Crossroads*, 26(1):14–17, September 2019. doi:10.1145/3355759.

18 Simon Perdrix and Philippe Jorrand. Classically controlled quantum computation. *Math. Struct. Comput. Sci.*, 16(04):601, July 2006. doi:10.1017/s096012950600538x.

19 John Preskill. Quantum Computing in the NISQ era and beyond. *Quantum*, 2:79, August 2018. doi:10.22331/q-2018-08-06-79.

20 Francisco Rios and Peter Selinger. A categorical model for a quantum circuit description language. In *Proc. of QPL*, volume 266, June 2017. doi:10.4204/EPTCS.266.11.

21 Neil Ross. *Algebraic and Logical Methods in Quantum Computation.* PhD thesis, Dalhousie University, 2015.

22 Peter Selinger. A brief survey of quantum programming languages. In *Proc. of FLOPS*, pages 1–6, 2004. doi:10.1007/978-3-540-24754-8_1.

23 Peter Selinger and Benoît Valiron. A lambda calculus for quantum computation with classical control. In *Proc. of TLCA*, pages 354–368, 2005. doi:10.1007/11417170_26.

24 Mingsheng Ying. *Foundations of Quantum Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2016. doi:10.1016/C2014-0-02660-3.

25 Bernhard Ömer. Classical concepts in quantum programming. *Int. J. Theor. Phys.*, 44(7):943–955, July 2005. doi:10.1007/s10773-005-7071-x.

A Type Derivations

$$\text{return}\,\dfrac{\text{var}\,\dfrac{}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_v a : \mathsf{Qubit}}}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_c^{\epsilon} \mathsf{return}\, a : \{\mathsf{Qubit}\}}$$

Figure 18 Sub-derivation Π', corresponding to the $u = 0$ branch of the lifted computational judgment $\_ : u\,\{\mathbb{1}\}\{\mathbb{1}\}, a : \mathsf{Qubit}; \emptyset \Vdash_c^{u\,\{\{\epsilon\}\}\{\{\epsilon\}\}} u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha, a)\} : u\,\{\{\mathsf{Qubit}\}\}\{\{\mathsf{Bit}\}\}$ required by sub-derivation Π of Figure 20.

$$\text{apply}\,\dfrac{\_ : \mathbb{1}; \emptyset \vdash_v Meas_\alpha : \mathsf{Circ}_\epsilon(\mathsf{Qubit}, \{\mathsf{Bit}\}) \qquad \text{var}\,\dfrac{}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_v a : \mathsf{Qubit}}}{\_ : \mathbb{1}, a : \mathsf{Qubit}; \emptyset \vdash_c^{\epsilon} \mathsf{apply}(Meas_\alpha, a) : \{\mathsf{Bit}\}}$$

Figure 19 Sub-derivation Π'', corresponding to the $u = 1$ branch of the lifted computational judgment $\_ : u\,\{\mathbb{1}\}\{\mathbb{1}\}, a : \mathsf{Qubit}; \emptyset \Vdash_c^{u\,\{\{\epsilon\}\}\{\{\epsilon\}\}} u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha, a)\} : u\,\{\{\mathsf{Qubit}\}\}\{\{\mathsf{Bit}\}\}$ required by sub-derivation Π of Figure 20.

$$\text{let}\,\dfrac{\text{apply}\,\dfrac{\emptyset; \emptyset \vdash_v ML : \mathsf{Circ}_{u\,\{\epsilon\}\{\epsilon\}}(\mathsf{Qubit}, u\,\{\mathbb{1}\}\{\mathbb{1}\}) \qquad \text{var}\,\dfrac{}{q : \mathsf{Qubit}; \emptyset \vdash_v q : \mathsf{Qubit}}}{\begin{array}{c} q : \mathsf{Qubit}; \emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{apply}_u(ML, q) : u\,\{\mathbb{1}\}\{\mathbb{1}\} \\ u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha, a)\} \in \mathcal{K}_{u\,\{\epsilon\}\{\epsilon\}}(TERM) \end{array}} \qquad \Pi' \quad \Pi''}{\begin{array}{l} q : \mathsf{Qubit}, a : \mathsf{Qubit}; \emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{let}\ \_ = \mathsf{apply}_u(ML, q)\ \mathsf{in} \\ \qquad\qquad\qquad\qquad u\,\{\mathsf{return}\,a\}\{\mathsf{apply}(Meas_\alpha, a)\} : u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\} \end{array}}$$

Figure 20 Sub-derivation Π, where the conclusion is the expansion of the (trivial) lifted computational judgment required by the *let* rule in Figure 21.

$$
\text{abs}\,\dfrac{
\text{return}\,\dfrac{
\text{abs}\,\dfrac{
\text{let}\,\dfrac{
\begin{array}{c}
\text{apply}\,\dfrac{\emptyset;\emptyset \vdash_v H : \mathsf{Circ}_\epsilon(\mathsf{Qubit},\{\mathsf{Qubit}\}) \qquad \text{var}\,\dfrac{}{q : \mathsf{Qubit};\emptyset \vdash_v q : \mathsf{Qubit}}}{q : \mathsf{Qubit};\emptyset \vdash_c^{\epsilon} \mathsf{apply}(H,q) : \{\mathsf{Qubit}\}} \qquad \Pi \\
\{\mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in}\ u\,\{\mathsf{return}\ a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} \in \mathcal{K}_\epsilon(TERM)
\end{array}
}{
\begin{array}{l}
a : \mathsf{Qubit}, q : \mathsf{Qubit};\emptyset \vdash_c^{u\,\{\epsilon\}\{\epsilon\}} \mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{ \\
\qquad \mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in} \\
\qquad u\,\{\mathsf{return}\ a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}
\end{array}
}
}{
\begin{array}{l}
q : \mathsf{Qubit};\emptyset \vdash_v \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{ \\
\qquad \mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in} \\
\qquad u\,\{\mathsf{return}\ a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} : \mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}
\end{array}
}
}{
\begin{array}{l}
q : \mathsf{Qubit};\emptyset \vdash_c^{\epsilon} \mathsf{return}\ \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{ \\
\qquad \mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in} \\
\qquad u\,\{\mathsf{return}\ a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} \\
\qquad : \{\mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}\}
\end{array}
}
}{
\begin{array}{l}
\emptyset;\emptyset \vdash_v \lambda q_{\mathsf{Qubit}}.\,\mathsf{return}\ \lambda a_{\mathsf{Qubit}}.\mathsf{let}\ q = \mathsf{apply}(H,q)\ \mathsf{in}\ \{ \\
\qquad \mathsf{let}\ \_ = \mathsf{apply}_u(ML,q)\ \mathsf{in} \\
\qquad u\,\{\mathsf{return}\ a\}\{\mathsf{apply}(Meas_\alpha,a)\}\} \\
\qquad : \mathsf{Qubit} \multimap \{\mathsf{Qubit} \multimap u\,\{\mathsf{Qubit}\}\{\mathsf{Bit}\}\}
\end{array}
}
$$

Figure 21 Type derivation for the Proto-Quipper-K program in Figure 14. For brevity, H, $Meas_\alpha$ and ML stand for the corresponding boxed circuits employed in Figure 14 and arrow annotations are omitted. Sub-derivation Π is given in Figure 20.

Expressing Ecumenical Systems in the $\lambda\Pi$-Calculus Modulo Theory

Emilie Grienenberger ✉ 🏠
Université Paris-Saclay, ENS Paris-Saclay, Inria, CNRS, Laboratoire Méthodes Formelles, France

Abstract

Systems in which classical and intuitionistic logics coexist are called ecumenical. Such a system allows for interoperability and hybridization between classical and constructive propositions and proofs. We study Ecumenical STT, a theory expressed in the logical framework of the λΠ-calculus modulo theory. We prove soudness and conservativity of four subtheories of Ecumenical STT with respect to constructive and classical predicate logic and simple type theory. We also prove the weak normalization of well-typed terms and thus the consistency of Ecumenical STT.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Proof theory; Theory of computation → Higher order logic; Theory of computation → Constructive mathematics

Keywords and phrases dependent types, predicate logic, higher order logic, constructivism, interoperability, ecumenical logics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.4

Acknowledgements The author thanks Gilles Dowek for his scientific advice and guidance, and the reviewers for their constructive and detailed returns.

1 Introduction

The λΠ-calculus modulo theory ($\lambda\Pi / \equiv$) [3] is a logical framework in which diverse systems – predicate logic, pure type systems [10], cumulative type systems [44], the ς-calculus [39], Matching Logic and more – can be expressed as theories. Using a common language to describe the logical foundations of various proof assistants allows more interoperability between the currently impermeable libraries of formal proofs. Indeed, it is a valuable tool in the design of translations [8, 43, 19, 24], the constitution of a common database of proofs [12], or even the hybridization of their proofs [9].

In the zoology of proof assistants, there are many examples of classical systems (the HOL family, PVS, etc) and constructive systems (Coq, Agda, Matita, etc). They rely on different sets of axioms: for instance, the axiom of the excluded-middle $\neg P \vee P$ is used in classical logic but not in intuitionistic logic [11]. These axioms define the *meaning* of logical symbols, thus constructive and classical disjunctions have different meanings. This observation does not only hold for disjunction and negation, but also for connectives that do not appear in the excluded-middle axiom. For example Peirce's formula $((P \Rightarrow Q) \Rightarrow Q) \Rightarrow P$, the equivalence $(\neg P \vee Q) \Leftrightarrow (P \Rightarrow Q)$ and the de Morgan laws hold classically but not intuitionistically. Using a unique symbol for two connectives with different significations is unsatisfactory, thus we can attempt to design logical systems where intuitionistic and classical symbols are written differently. Such logical systems are called *ecumenical* [38, 36].

An ecumenical expression of logic and of mathematics has many advantages. First, it allows to use the expressivity of classical logic while explicitely keeping constructive properties. For example, a program can be extracted from a proof of $\forall x.\ \exists y.\ S(x, y)$ – where ∀ and ∃ are intuitionistic – even if the specification $S(x, y)$ is classical, reflecting the fact that an algorithm can both be effective and have a classical correctness proof. Second, intuitionistic and classical proofs coexisting in the same logical system can be stored in a common database of formal proofs, while using two separate logical systems entails two separate databases – or the loss of readily available constructive information.

In [6] is introduced *theory* $\mathcal{U}$, which is a $\lambda\Pi / \equiv$ theory in which all proofs of minimal, constructive, classical, and ecumenical predicate logic, minimal, constructive, classical, and ecumenical simple type theory with or without prenex polymorphism or predicate subtyping, and the calculus of constructions be can expressed. More precisely, [6] includes a presentation of the axioms of theory $\mathcal{U}$ and a proof of well-typedness and modularity of the constructed theory. Many axioms and fragments of theory $\mathcal{U}$ have been studied in isolation [26, 43, 13, 3], however some of these studies lack proofs of normalization, consistency, soundness or conservativity with respect to appropriate reference systems. Some axioms of theory $\mathcal{U}$ have not been studied together; notably, there is currently no proof of normalization or consistency for the whole theory. In this paper, we study the ecumenical subtheory of theory $\mathcal{U}$, called Ecumenical STT, by reviewing existing results and establishing its soundness, conservativity, normalization, and consistency.

Related work

Examples of first order ecumenical systems are found in sequent calculi [34, 22, 14, 33, 37] and natural deduction [38, 36]. Some rely on double negation translations to define their classical connectives and rules, as in [14] and to a lesser extent [38]. Another point of difference is the way predicates and atoms are handled. In [38], there is a classical and an intuitionistic copy of each predicate symbol. The system described in [14] avoids this split by relying on total provability – the equivalence of provability in its classical fragment and in LK is valid for sequents with empty contexts only. Ecumenical STT does not rely on copies of predicates nor on total provability, as these features seem unsuitable for the specific purpose of interoperability between proof systems. To our knowledge, none of the aforementioned ecumenical systems are extended to the higher order.

The termination of rewriting systems being an important problem in logic and software verification, there is a multitude of dedicated theoretical results and automatized tools. Many are specific to first order rewriting [21, 32], or simply-typed theories [31, 5]. A few are designed for higher order and/or dependently typed systems [30, 28, 7], which is the framework of this paper. The existing tools using dependency pairs are currently unable to conclude to the normalization of Ecumenical STT, thus we base our normalization proof on models of $\lambda\Pi / \equiv$ theories in variants of pre-Heyting algebras developed in [15].

Normalization is also a valuable, if not crucial tool to establish the conservativity of $\lambda\Pi / \equiv$ theories as shown in [10, 2]. The conservativity proofs lead in this paper use the framework these aforementioned studies provide.

Outline

In Section 2 is presented the logical framework $\lambda\Pi$-calculus modulo theory ($\lambda\Pi / \equiv$) and the $\lambda\Pi / \equiv$ theory of Ecumenical STT [6]. Some meta-theoretical properties of Ecumenical STT are discussed in Section 3; notably we establish its weak normalization and the decidability of type-checking in Section 3.2. Finally, the soundness and conservativity with respect to appropriate reference systems – constructive and classical predicate logic and simple-type theory – of four subtheories of Ecumenical STT are proven in Section 4 and Section 5. We conclude as to the consistency of Ecumenical STT in Section 5.3.

2 Expressing ecumenism in $\lambda\Pi$-calculus modulo theory

2.1 The $\lambda\Pi$-calculus modulo theory

The logical framework $\lambda\Pi$-calculus modulo theory ($\lambda\Pi/\equiv$) [3], based on the Edinburgh Logical Framework (LF) [25], is an extension of simply-typed lambda calculus with dependent types and a primitive notion of computation via the definition of rewrite rules [42]. Formally, $\lambda\Pi/\equiv$ *terms* are defined inductively by

$$t, u, \ldots = \texttt{TYPE} \mid \texttt{KIND} \mid x \mid c \mid \lambda x : t.\, u \mid t\ u \mid (x : t) \to u$$

where x belongs to an infinite set of variables $\mathcal{V}$ and c belongs to a finite or infinite set of constants $\mathcal{C}$. The terms TYPE and KIND are respectively the type of $\lambda\Pi/\equiv$ types, and the type of *kinds*, that is of $\lambda\Pi/\equiv$ type families; both terms are called *sorts* and often denoted by s. Dependent products, *i.e.* terms of the form $(x : t) \to u$, allow the definition of indexed type families. For example, the family of vectors indexed by their length $n \in \mathbb{N}$ can be defined by declaring $\texttt{Vector} : (n : \mathbb{N}) \to \texttt{TYPE}$; in this context a vector of length 3 can be declared by $\texttt{v} : \texttt{Vector}\ 3$. In the pathological case where the variable x does not appear free in term u, we write $(x : t) \to u$ as a simple product $t \to u$. We denote by $fv(t)$ the set of free variables of a term t, and $(u/x)t$ the substitution of variable x by a term u in a term t.

A $\lambda\Pi/\equiv$ *theory* is defined conjointly by a finite set of declarations Σ and a finite set of rewrite rules $\mathcal{R}$. A *declaration* is the assignment of a type T to a constant $c \in \mathcal{C}$, denoted by $c : T$. We denote respectively by $const(\Sigma)$ and $\Lambda(\Sigma)$ the set of constants assigned in signature Σ and the set of terms written with the set $const(\Sigma)$ of constants. A *rewrite rule* is a pair of terms $\ell \hookrightarrow r$ such that $\ell = c\ t_1\ \ldots\ t_n$ where c is a constant and $t_1, \ldots, t_n$ are terms. For example, the rewrite rule $\texttt{Vector}\ (\texttt{n} + \texttt{1}) \hookrightarrow \texttt{NonEmptyVector}$ assimilates all vector types of strictly positive length to a constant type NonEmptyVector.

We denote by $\hookrightarrow_\beta$ the β-reduction. Given a set of rewrite rules $\mathcal{R}$, the relation $\hookrightarrow_\mathcal{R}$ denotes the smallest relation closed by term constructors (λ-abstraction, application and dependent product) and substitution containing $\mathcal{R}$. Finally, we write $\hookrightarrow_{\beta\mathcal{R}}$ for the union $\hookrightarrow_\beta \cup \hookrightarrow_\mathcal{R}$ and $\equiv_{\beta\mathcal{R}}$ for the smallest equivalence relation containing $\hookrightarrow_{\beta\mathcal{R}}$.

Proofs in $\lambda\Pi/\equiv$ are similar to LF proofs. However, the *conversion* rule allows to assimilate types modulo $\equiv_{\beta\mathcal{R}}$ and not only modulo $\equiv_\beta$. Formally, *typing contexts* are finite sets of variable assignments of the form $x : T$ and are denoted in the following by Γ or Δ; the empty context is written $[]$. Typing *judgments* are written "$\vdash_{\Sigma,\mathcal{R}} \Gamma$ wf" and "$\Gamma \vdash_{\Sigma,\mathcal{R}} t : T$", where Γ is a typing context, t and T are terms of $\Lambda(\Sigma)$, and "wf" stands for *well-formed*. The typing rules of $\lambda\Pi/\equiv$ in a theory Σ, $\mathcal{R}$ are represented in Figure 1.

2.2 Ecumenical STT and its subtheories

The following section describes the expression of first and higher order ecumenism in theory $\mathcal{U}$ [6]. All associated declarations and rewrite rules are represented in Figure 2.

On Figure 2a is represented the base of the encoding. The type Set is the type of the object-types of the encoded theories (for example the sorts of predicate logic and simple types of STT). The symbol El embeds object-types into $\lambda\Pi/\equiv$ types: an object-type T and any one of its elements t can be manipulated as $\lambda\Pi/\equiv$ objects while still being linked by the typing relation $t : \text{El}\ T$. The type of propositions Prop and the embedding Prf of propositions into the type of their proofs are similarly defined.

On Figure 2c are defined the constructive connectives and quantifiers of theory $\mathcal{U}$; more precisely they are declared and then defined by rewriting. As an example, the definition of the implication is based on the Curry-De-Bruijn-Howard correspondance: the proofs of

$$\frac{}{\vdash_{\Sigma,\mathfrak{R}} [\,]\ \mathrm{wf}}\ (\mathrm{empty}) \qquad \frac{\Gamma \vdash_{\Sigma,\mathfrak{R}} A : s}{\vdash_{\Sigma,\mathfrak{R}} \Gamma, x : A\ \mathrm{wf}}\ (\mathrm{decl}) \qquad \frac{\vdash_{\Sigma,\mathfrak{R}} \Gamma\ \mathrm{wf}}{\Gamma \vdash_{\Sigma,\mathfrak{R}} \mathtt{TYPE} : \mathtt{KIND}}\ (\mathrm{sort})$$

$$\frac{\vdash_{\Sigma,\mathfrak{R}} \Gamma\ \mathrm{wf} \qquad x : A \in \Gamma}{\Gamma \vdash_{\Sigma,\mathfrak{R}} x : A}\ (\mathrm{var}) \qquad \frac{\vdash_{\Sigma,\mathfrak{R}} \Gamma\ \mathrm{wf} \qquad \vdash_{\Sigma,\mathfrak{R}} A : s \qquad c : A \in \Sigma}{\Gamma \vdash_{\Sigma,\mathfrak{R}} c : A}\ (\mathrm{const})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathfrak{R}} A : \mathtt{TYPE} \qquad \Gamma, x : A \vdash_{\Sigma,\mathfrak{R}} B : s}{\Gamma \vdash_{\Sigma,\mathfrak{R}} (x : A) \to B : s}\ (\mathrm{prod})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathfrak{R}} A : \mathtt{TYPE} \qquad \Gamma, x : A \vdash_{\Sigma,\mathfrak{R}} B : s \qquad \Gamma, x : A \vdash_{\Sigma,\mathfrak{R}} t : B}{\Gamma \vdash_{\Sigma,\mathfrak{R}} \lambda x : A.\, t : (x : A) \to B}\ (\mathrm{abs})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathfrak{R}} t : (x : A) \to B \qquad \Gamma \vdash_{\Sigma,\mathfrak{R}} u : A}{\Gamma \vdash_{\Sigma,\mathfrak{R}} t\ u : (u/x)B}\ (\mathrm{app})$$

$$\frac{\Gamma \vdash_{\Sigma,\mathfrak{R}} t : A \qquad \Gamma \vdash_{\Sigma,\mathfrak{R}} B : s \qquad A \equiv_{\beta\mathfrak{R}} B}{\Gamma \vdash_{\Sigma,\mathfrak{R}} t : B}\ (\mathrm{conv})$$

Figure 1 Typing rules of $\lambda\Pi / \equiv$ in theory $\Sigma, \mathfrak{R}$.

$A \Rightarrow B$ are functions from proofs of A to proofs of B. We use a rewrite rule to identify the corresponding types $\mathrm{Prf}(A \Rightarrow B)$ and $\mathrm{Prf}(A) \to \mathrm{Prf}(B)$. All other connectives and quantifiers are defined *à la Russel*, mimicking the elimination rules of natural deduction; once again the relevant types are identified by rewriting. Note that the quantifiers $\forall$ and $\exists$ bind a variable from a sort in Set: we define them as taking as arguments an object-type $a : \mathrm{Set}$ and a function from elements of a to propositions. We call *Constructive Predicate Logic* the $\lambda\Pi / \equiv$ theory formed by

- $\Sigma^c_{FO} = \{\, (C\text{-decl}) : C \in \{\, \mathrm{Set}, \mathrm{El}, \iota, \mathrm{Prop}, \top, \bot, \neg, \wedge, \vee, \forall, \exists \,\}\,\}$
- and $\mathfrak{R}^c_{FO} = \{\, (C\text{-red}) : C \in \{\, \top, \bot, \neg, \wedge, \vee, \forall, \exists \,\}\,\}$.

In Figure 2d are defined the classical connectives and quantifiers, using their constructive counterpart and double negations, which is a frequent strategy to build ecumenical logics. We begin by introducing a classical version of Prf, enabling us to add prenex double negations whenever necessary. This definition has many advantages: we are able to add double negations to isolated atomic formulas, which is a recurrent problem in the design of ecumenical systems, without adding too much heaviness to our system. The classical connectives and quantifiers are now defined using their constructive counterpart and internal double negations.

We call *Ecumenical Predicate Logic* the $\lambda\Pi / \equiv$ theory formed by $\Sigma^e_{FO} = \Sigma^c_{FO} \cup \{\, (C\text{-decl}) : C \in \{\, \mathrm{Prf}_c, \wedge_c, \vee_c, \forall_c, \exists_c \,\}\,\}$ and $\mathfrak{R}^e_{FO} = \mathfrak{R}^c_{FO} \cup \{\, (C\text{-red}) : C \in \{\, \mathrm{Prf}_c, \wedge_c, \vee_c, \forall_c, \exists_c \,\}\,\}$.

Simple type theory (STT) can be expressed either as a first order theory or as an extension of predicate logic. To avoid nestling multiple encodings, we choose the latter option, as shown in Figure 2b. Predicate Logic is extended with the object type of propositions o and the function type arrow $\leadsto$. These definitions allow to construct simple types and assimilate propositions to objects.

We respectively call *Constructive STT* and *Ecumenical STT* the $\lambda\Pi / \equiv$ theories $\Sigma^c_{HO} = \Sigma^c_{FO} \cup \{\, (C\text{-decl}) : C \in \{\, o, \leadsto \,\}\,\}, \mathfrak{R}^c_{HO} = \mathfrak{R}^c_{FO} \cup \{\, (C\text{-red}) : C \in \{\, o, \leadsto \,\}\,\}$ and $\Sigma^e_{HO} = \Sigma^e_{FO} \cup \Sigma^c_{HO}, \mathfrak{R}^e_{HO} = \mathfrak{R}^e_{FO} \cup \mathfrak{R}^c_{HO}$. For readability purposes, we respectively denote by $\vdash^c_{HO}$ and $\vdash^e_{HO}$ the provability relations $\vdash_{\Sigma^c_{HO}, \mathfrak{R}^c_{HO}}$ of Constructive STT and $\vdash_{\Sigma^e_{HO}, \mathfrak{R}^e_{HO}}$ of Ecumenical STT.

(Set-decl)	$\mathsf{Set} : \mathtt{TYPE}$
(ι-decl)	$\iota : \mathsf{Set}$
(El-decl)	$\mathsf{El} : \mathsf{Set} \to \mathtt{TYPE}$
(Prop-decl)	$\mathsf{Prop} : \mathtt{TYPE}$
(Prf-decl)	$\mathsf{Prf} : \mathsf{Prop} \to \mathtt{TYPE}$

(a) Base of the encoding.

(o-decl)	$o : \mathsf{Set}$
(o-red)	$\mathsf{El}\ o \hookrightarrow \mathsf{Prop}$
($\leadsto$-decl)	$\leadsto : \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}$
($\leadsto$-red)	$\mathsf{El}\ (x \leadsto y) \hookrightarrow \mathsf{El}\ x \to \mathsf{El}\ y$

(b) Higher order.

($\top$-decl)	$\top : \mathsf{Prop}$
($\top$-red)	$\mathsf{Prf}\ \top \hookrightarrow (z : \mathsf{Prop}) \to \mathsf{Prf}\ z \to \mathsf{Prf}\ z$
($\bot$-decl)	$\bot : \mathsf{Prop}$
($\bot$-red)	$\mathsf{Prf}\ \bot \hookrightarrow (z : \mathsf{Prop}) \to \mathsf{Prf}\ z$
($\Rightarrow$-decl)	$\Rightarrow : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\Rightarrow$-red)	$\mathsf{Prf}\ x \Rightarrow y \hookrightarrow \mathsf{Prf}\ x \to \mathsf{Prf}\ y$
($\neg$-decl)	$\neg : \mathsf{Prop} \to \mathsf{Prop}$
($\neg$-red)	$\mathsf{Prf}\ (\neg\, x) \hookrightarrow \mathsf{Prf}\ x \to (z : \mathsf{Prop}) \to \mathsf{Prf}\ z$
($\wedge$-decl)	$\wedge : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\wedge$-red)	$\mathsf{Prf}\ (x \wedge y) \hookrightarrow (z : \mathsf{Prop}) \to (\mathsf{Prf}\ x \to \mathsf{Prf}\ y \to \mathsf{Prf}\ z) \to \mathsf{Prf}\ z$
($\vee$-decl)	$\vee : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\vee$-red)	$\mathsf{Prf}\ (x \vee y) \hookrightarrow (z : \mathsf{Prop}) \to (\mathsf{Prf}\ x \to \mathsf{Prf}\ z) \to (\mathsf{Prf}\ y \to \mathsf{Prf}\ z) \to \mathsf{Prf}\ z$
($\forall$-decl)	$\forall : (a : \mathsf{Set}) \to (\mathsf{El}\ a \to \mathsf{Prop}) \to \mathsf{Prop}$
($\forall$-red)	$\mathsf{Prf}\ (\forall\ a\ p) \hookrightarrow (z : \mathsf{El}\ a) \to \mathsf{Prf}\ (p\ z)$
($\exists$-decl)	$\exists : (a : \mathsf{Set}) \to (\mathsf{El}\ a \to \mathsf{Prop}) \to \mathsf{Prop}$
($\exists$-red)	$\mathsf{Prf}\ (\exists\ a\ p) \hookrightarrow (z : \mathsf{Prop}) \to ((x : \mathsf{El}\ a) \to \mathsf{Prf}\ (p\ x) \to \mathsf{Prf}\ z) \to \mathsf{Prf}\ z$

(c) Constructive connectives and quantifiers.

(Prf_c-decl)	$\mathsf{Prf}_c : \mathsf{Prop} \to \mathtt{TYPE}$
(Prf_c-red)	$\mathsf{Prf}_c \hookrightarrow \lambda x : \mathsf{Prop}.\, \mathsf{Prf}\ (\neg\neg\, x)$
($\Rightarrow_c$-decl)	$\Rightarrow_c : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\Rightarrow_c$-red)	$\Rightarrow_c \hookrightarrow \lambda x : \mathsf{Prop}.\, [\lambda y : \mathsf{Prop}.\, (\neg\neg\, x) \Rightarrow_c (\neg\neg\, y)]$
($\wedge_c$-decl)	$\wedge_c : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\wedge_c$-red)	$\wedge_c \hookrightarrow \lambda x : \mathsf{Prop}.\, [\lambda y : \mathsf{Prop}.\, (\neg\neg\, x) \wedge (\neg\neg\, y)]$
($\vee_c$-decl)	$\vee_c : \mathsf{Prop} \to \mathsf{Prop} \to \mathsf{Prop}$
($\vee_c$-red)	$\vee_c \hookrightarrow \lambda x : \mathsf{Prop}.\, [\lambda y : \mathsf{Prop}.\, (\neg\neg\, x) \vee (\neg\neg\, y)]$
($\forall_c$-decl)	$\forall_c : (a : \mathsf{Set}) \to (\mathsf{El}\ a \to \mathsf{Prop}) \to \mathsf{Prop}$
($\forall_c$-red)	$\forall_c \hookrightarrow \lambda a : \mathsf{Set}.\, [\lambda p : \mathsf{El}\ a \to \mathsf{Prop}.\, \forall\ a\ (\lambda x : \mathsf{El}\ a.\, \neg\neg(p\ x))]$
($\exists_c$-decl)	$\exists_c : (a : \mathsf{Set}) \to (\mathsf{El}\ a \to \mathsf{Prop}) \to \mathsf{Prop}$
($\exists_c$-red)	$\exists_c \hookrightarrow \lambda a : \mathsf{Set}.\, [\lambda p : \mathsf{El}\ a \to \mathsf{Prop}.\, \exists\ a\ (\lambda x : \mathsf{El}\ a.\, \neg\neg(p\ x))]$

(d) Classical connectives and quantifiers.

Figure 2 Definition of Ecumenical STT in $\lambda\Pi / \equiv$.

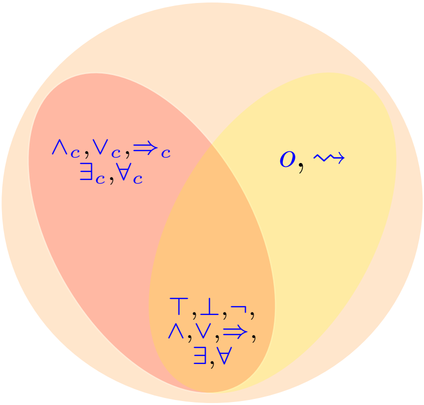


Figure 3 Logical fragments of theory $\mathcal{U}$.

We call the set of these four theories, represented in Figure 3, the "logical fragments of theory $\mathcal{U}$".

In Ecumenical Predicate Logic and Ecumenical STT, hybrid propositions and proofs can be expressed, for example in context $P : \mathrm{Prop}, Q : \mathrm{Prop}$, one can prove that $\mathrm{Prf}((P \wedge Q) \Rightarrow_c P)$ is inhabited and that $\mathrm{Prf}((P \wedge_c Q) \Rightarrow P)$ is not.

3 Properties of the logical fragments of theory $\mathcal{U}$

The system $\lambda\Pi / \equiv$ is very permissive due to the minimal restrictions on the user-defined rewriting system $\mathcal{R}$. In general, there is no guarantee of properties such as subject reduction, type uniqueness, or the decidability of type-checking [40, 2]. To ensure that the theories defined in the previous section are well-behaved, further properties need to be established such as the confluence, well-typedness, and normalization of the associated rewriting system.

3.1 Well-typedness

A $\lambda\Pi / \equiv$ theory $\Sigma, \mathcal{R}$ is said to be *well-typed* if

1. the rewriting system $\hookrightarrow_{\beta\mathcal{R}}$ is *confluent* [42, 40, Definition 1.1.5.],
2. for every declaration $c : T$ in signature Σ, term T is typed by a sort s in theory $\Sigma, \mathcal{R}$,
3. for every rule $\ell \hookrightarrow r$ in $\mathcal{R}$, typing context Γ, type T, and substitution σ such that $\Gamma \vdash_{\Sigma,\mathcal{R}} \sigma\ell : T$, then $\Gamma \vdash_{\Sigma,\mathcal{R}} \sigma r : T$.

Item 3 ensures that $\hookrightarrow_{\mathcal{R}}$ enjoys *subject reduction* [40, Definition 2.4.4], *i.e.* $\Gamma \vdash_{\Sigma,\mathcal{R}} t : T$ and $t \hookrightarrow_{\mathcal{R}} u$ implies $\Gamma \vdash_{\Sigma,\mathcal{R}} u : T$ for any Γ, t, u, T. Item 1 ensures that the product is *injective* in theory $\Sigma, \mathcal{R}$ [4], *i.e.* $(x : t_1) \to u_1 \equiv_{\beta\mathcal{R}} (x : t_2) \to u_2$ implies $t_1 \equiv_{\beta\mathcal{R}} t_2$ and $u_1 \equiv_{\beta\mathcal{R}} u_2$ for any x, t_1, t_2, u_1, u_2. These properties guarantee in turn that $\hookrightarrow_\beta$ preserves typing. Note that the injectivity of the product is also called *product compatibility* in the literature [40, 44].

The well-typedness of all fragments of theory $\mathcal{U}$, including the four logical fragments studied in this paper, is established in [6]. Alternatively, the framework of *strongly well-formed* $\lambda\Pi / \equiv$ theories [40] could also be used in the specific case of the logical fragments to ensure well-typedness. This framework is not sufficient for the entirety of theory $\mathcal{U}$, as fragments of theory $\mathcal{U}$ that include the definition of *predicate subtyping* [26] are not strongly well-formed.

▶ **Lemma 1** (Well-typedness [6, Theorem 9]). *All logical fragments of theory $\mathcal{U}$ are well-typed.*

▶ **Corollary 2** (Subject reduction). *Let $\Sigma, \mathscr{R}$ be one of the four logical fragments of theory $\mathcal{U}$. Let t, u, T be terms of $\Lambda(\Sigma)$ and Γ a typing context such that $\Gamma \vdash_{\Sigma,\mathscr{R}} t : T$ and $t \hookrightarrow_{\beta\mathscr{R}} u$, then $\Gamma \vdash_{\Sigma,\mathscr{R}} u : T$.*

▶ **Corollary 3** (Fragment theorem [6, Theorem 7]). *Let $\Sigma_1, \mathscr{R}_1$ and $\Sigma_2, \mathscr{R}_2$ be two of the four logical fragments of theory $\mathcal{U}$ such that $\Sigma_1 \subseteq \Sigma_2$. Let t, T be terms of $\Lambda(\Sigma_1)$ and Γ a typing context such that codom$(\Gamma) \subset \Lambda(\Sigma_1)$. If $\Gamma \vdash_{\Sigma_2,\mathscr{R}_2} t : T$, then $\Gamma \vdash_{\Sigma_1,\mathscr{R}_1} t : T$.*

3.2 Normalization

A sufficient condition for the decidability of type-checking in a well-typed theory $\Sigma, \mathscr{R}$ is the *normalization* [42] of the rewriting system $\hookrightarrow_{\beta\mathscr{R}}$. Indeed, using a normalization strategy and the confluence of $\hookrightarrow_{\beta\mathscr{R}}$, the convertibility modulo $\equiv_{\beta\mathscr{R}}$ of two terms $A, B \in \Lambda(\Sigma)$ is decidable by computing the normal forms of A and B and testing if they are equal (up to α-renaming). In this case, the applicability of the conversion rule (conv) is decidable, ensuring in turn the decidability of type-checking modulo $\Sigma, \mathscr{R}$. Normalization is also a first step towards a proof of consistency for a given theory.

The strong normalization of β-reduction is established for the $\lambda\Pi$-calculus [25], and the additional rewriting systems defined by the four logical fragments of theory $\mathcal{U}$ are obviously normalizing. However, weak and strong normalization are not modular in higher-order rewriting settings [1]; as a consequence, the normalization of $\hookrightarrow_{\beta\mathscr{R}}$ cannot be deduced from the fact that $\hookrightarrow_{\beta}$ and $\hookrightarrow_{\mathscr{R}}$ are both normalizing.

Some theoretical results and tools have been developed in order to prove normalization of term rewriting systems in dependent type theories [20, 35, 7]. However, these results cannot be directly used to prove the normalization of the logical fragments of $\mathcal{U}$, and more generally of theory $\mathcal{U}$. The rule ($\rightsquigarrow$-red) is one of the many problematic rules: the right-hand side is a product, thus the rule is not *arity-preserving* [18], and is pinpointed by SizeChangeTool [7] as being self-looping. As a consequence, the consistency and type-checking decidability of theory $\mathcal{U}$ and many of its fragments is still an open question.

In the following sections, we begin to answer this question by proving the weak normalization of Ecumenical STT, from which the decidability of type-checking and consistency will ensue. The following normalization proof relies on a notion of *models* of $\lambda\Pi / \equiv$ theories valued in structures named Π-*algebras*, which are similar to pre-Heyting algebras [27, 17]. Any $\lambda\Pi / \equiv$ theory which admits a model in every full ordered and complete Π-algebra is called *super-consistent*. In [15], the author establishes the strong normalization of $\hookrightarrow_{\beta}$ over well-typed terms for any super-consistent theory using *reducibility candidates* [41, 23]. Moreover, the super-consistency of an expression of minimal STT with parametric quantifiers is proven. Sections 3.2.1 and 3.2.2 extend the models of minimal STT with parametric quantifiers described in [15] to Constructive STT, thus proving that Constructive STT is super-consistent. Finally, the strong normalization of $\hookrightarrow_{\beta}$ over $\lambda\Pi / \equiv$ terms well-typed in Constructive STT is used to prove weak normalization for all logical fragments of theory $\mathcal{U}$.

3.2.1 Super-consistency

In the following section, we define all the notions necessary to state and prove the super-consistency of Constructive STT.

▶ **Definition 4.** *A* full, complete, and ordered Π-algebra *is formed with*

- *a preordered set $(\mathcal{B}, \leq)$ with a maximal element $\tilde{\top}$ of $\mathcal{B}$,*
- *a function $\tilde{\wedge} : \mathcal{B} \times \mathcal{B} \to \mathcal{B}$ such that $a \mathbin{\tilde{\wedge}} b$ is a greatest lower bound of $\{ a, b \}$ for $\leq$,*

- *a function* $\tilde{\Pi} : \mathcal{B} \times (\mathcal{P}(\mathcal{B}) \setminus \emptyset) \to \mathcal{B}$ *such that* $a \leq \tilde{\Pi}(b, S)$ *if for all* $c \in S$, $a \mathbin{\tilde{\wedge}} b \leq c$,
- *and an order relation* $\sqsubseteq$ *over* $\mathcal{B}$ *with respect to which* $\tilde{\Pi}$ *is left anti-monotonic and right monotonic, and for which every subset of* $\mathcal{B}$ *has a least upper bound.*

Note that relations $\leq$ and $\sqsubseteq$ need not be in any way related.

▶ **Definition 5.** *A* model valued in a full, ordered, and complete Π-algebra *is a triplet of interpretation functions* $(\llbracket\cdot\rrbracket^i)_{1\leq i\leq 3}$ *and a set* $\mathcal{V}$. *The* i^{th} *interpretation* $\llbracket\cdot\rrbracket^i$ *takes as arguments a term* t *and* $i-1$ *variable assignments* $(\phi_j)_{1\leq j<i}$ *such that* $fv(t) \subseteq \bigcap_{1\leq j<i} dom(\phi_j)$ *and returns a value* $\llbracket t\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} \in \mathcal{V}$ *such that*

- $\llbracket \mathit{KIND}\rrbracket^2_{\phi_1} = \llbracket \mathit{TYPE}\rrbracket^2_{\phi_1} = \mathcal{B}$ *and* $\llbracket \mathit{KIND}\rrbracket^3_{\phi_1,\phi_2} = \llbracket \mathit{TYPE}\rrbracket^3_{\phi_1,\phi_2} = \tilde{\top}$,
- $\llbracket (x:t) \to u\rrbracket^3_{\phi_1,\phi_2} = \tilde{\Pi}(\llbracket t\rrbracket^3_{\phi_1,\phi_2}, \{\, \llbracket u\rrbracket^3_{(\phi_1,x=c_1),(\phi_2,x=c_2)} : c_1 \in \llbracket t\rrbracket^1, c_2 \in \llbracket t\rrbracket^2_{\phi_1} \,\})$

This definition is a simplification of its counterpart in [15], which allows models with an arbitrary number of interpretation functions. However, only three levels of interpretation are required to show the super-consistency of Constructive STT.

▶ **Definition 6** (Compatibility). *Let* $(\llbracket\cdot\rrbracket_i)_{1\leq i\leq 3}$ *be a model valued in a full, ordered, and complete* Π*-algebra* $\mathcal{B}$ *and* $i \in \{\,1, 2, 3\,\}$. *The variable assignments* $\phi_1, \ldots, \phi_{i-1}$ *are* compatible *with a typing context* Δ *if for all* $1 \leq j < i$ *and* $(x : A) \in \Delta$ *we have* $\phi_j(x) \in \llbracket A\rrbracket^j_{\phi_1,\ldots,\phi_{j-1}}$.

▶ **Definition 7** (Model of a theory). *A model valued in a full, ordered, and complete* Π*-algebra* $\mathcal{B}$ *is a model of a theory* $\Sigma, \mathcal{R}$ *if and only if the following conditions are met for every* $i \in \{\,1, 2, 3\,\}$*, typing context* Δ*, terms* $t, u, A, B, C \in \Lambda(\Sigma)$*, variables* $x, y \in \mathcal{V}$*, and compatible variable assignments* $\phi_1, \ldots, \phi_{i-1}$*:*

Variable assignment: *if* $i \geq 2$*, then for every* $(x : A) \in \Delta$*, we have* $\llbracket x\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} = \phi_{i-1}(x)$.

Well-typedness: *if* $i \geq 2$ *and* $\Delta \vdash_{\Sigma,\mathcal{R}} t : A$*, then* $\llbracket t\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} \in \llbracket A\rrbracket^{i-1}_{\phi_1,\ldots,\phi_{i-2}}$.

Weakening: *if* $\Delta \vdash_{\Sigma,\mathcal{R}} t : A$ *and* $y \notin dom(\Delta)$*, then* $\llbracket t\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} = \llbracket t\rrbracket^i_{(\phi_1,y=a_1),\ldots,(\phi_{i-1},y=a_{n-1})}$.

Substitution: *if* $\Delta(y : C) \vdash_{\Sigma,\mathcal{R}} t : B$, $\Delta \vdash_{\Sigma,\mathcal{R}} u : C$*, then*
$\llbracket (u/y)t\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} = \llbracket t\rrbracket^i_{(\phi_1,y=\llbracket u\rrbracket^2_{\phi_1}),\ldots,(\phi_{i-1},y=\llbracket u\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}})}$.

Validity of the congruence: *if* $\Delta \vdash_{\Sigma,\mathcal{R}} A : C$, $\Delta \vdash_{\Sigma,\mathcal{R}} B : C$*, and* $A \equiv_{\beta\mathcal{R}} B$*, then*
$\llbracket A\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}} = \llbracket B\rrbracket^i_{\phi_1,\ldots,\phi_{i-1}}$

Validity of the axioms: *if* $(c : A) \in \Gamma$*, then* $\llbracket A\rrbracket^3 \geq \tilde{\top}$.

▶ **Definition 8** (Super-consistency). *A* $\lambda\Pi / \equiv$ *theory* $\Sigma, \mathcal{R}$ *admitting a model valued in every full, ordered, and complete* Π*-algebra is called* super-consistent.

▶ **Theorem 9** ([15, Theorem 5.1]). $\to_\beta$ *is strongly normalizing over well-typed terms in super-consistent* $\lambda\Pi / \equiv$ *theories.*

Note that this result does not yield normalization for the whole rewriting system $\hookrightarrow_{\beta\mathcal{R}}$. This theorem is more generally applicable to non-terminating user-defined rewriting systems $\hookrightarrow_{\mathcal{R}}$, which is useful when $\equiv_{\beta\mathcal{R}}$ is a decidable congruence but $\hookrightarrow_{\mathcal{R}}$ is non-terminating. A trivial example would be $\mathcal{R} = \{\, x \hookrightarrow x \,\}$.

▶ **Theorem 10** ([15, Theorem 4.3]). *Minimal STT is super-consistent.*

3.2.2 Models of Constructive STT

In this section, we extend the model of Minimal STT valued in a given arbitrary full, ordered, and complete Π-algebra $\mathcal{B}$ defined in [15] to a model of Ecumenical STT.

3.2.2.1 Model of Minimal STT

Let $\{ e \}$ be an arbitrary one-element set and $\mathcal{A}$ the smallest set containing $\mathcal{B}$ and $\{ e \}$, and closed by cartesian product (denoted $\times$) and exponentiation (denoted $\mathcal{F}$). Let $\mathcal{V}$ be the smallest set containing $\mathcal{A}$, $\mathcal{B}$, and $\{ e \}$ and closed by cartesian product and *dependent function space*, *i.e.* if S is an element of $\mathcal{V}$ and T a family of elements of $\mathcal{V}$ indexed by S, then the set $\mathcal{F}_d(S, T)$ of functions mapping each element s of S to an element of T_s is in $\mathcal{V}$. We define the interpretation functions $(\llbracket\cdot\rrbracket^i)_{1 \leq i \leq 3}$ as described in Figure 4.

▶ **Lemma 11** ([15, Theorem 4.2]). *The model $(\llbracket\cdot\rrbracket^i)_{1 \leq i \leq 3}$ is a model of Minimal STT in $\mathcal{B}$.*

3.2.2.2 Extension of the model to Constructive STT

The three interpretation functions of the model defined in Figure 4 are extended to define a model of Constructive STT valued in the given Π-algebra $\mathcal{B}$ with the following method:

1. we extend the model to interpret the constructive connectives $\top$, $\bot$, $\neg$, $\exists$, $\wedge$, and $\vee$;
2. we prove that the typing of these connectives is preserved by these interpretations;
3. we prove that the axioms declaring these connectives, *i.e.* ($\top$-decl), ($\bot$-decl), ($\neg$-decl), ($\exists$-decl), ($\wedge$-decl), and ($\vee$-decl), are valid in this extension.
4. we prove that reduction by the rewrite rules defining these connectives, *i.e.* ($\top$-red), ($\bot$-red), ($\neg$-red), ($\exists$-red), ($\wedge$-red), and ($\vee$-red), leaves the interpretations invariant.

These verifications ensure that our extension is a model of Constructive STT. The most problematic step of this method is Item 4. As an example, constructive conjunction is not directly defined by a rewrite rule of the form $\wedge \hookrightarrow t$ or $a \wedge b \hookrightarrow t$, but by a rule of the form $\mathrm{Prf}\ (a \wedge b) \hookrightarrow t$ which needs to preserve interpretation. As a consequence, the interpretation of Prf and of the application need to be taken into account while accomplishing Item 1.

For example, as $\mathrm{Prf}\ \top \hookrightarrow (z : \mathrm{Prop}) \to \mathrm{Prf}\ z \to \mathrm{Prf}\ z$, we need to define $\llbracket \top \rrbracket^3_{\phi,\psi}$ such that $\llbracket \mathrm{Prf}\ \top \rrbracket^3_{\phi,\psi} = \llbracket (z : \mathrm{Prop}) \to \mathrm{Prf}\ z \to \mathrm{Prf}\ z \rrbracket^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{ \tilde{\Pi}(c, \{ c \}) \ :\ c \in \mathcal{B} \})$; here we conclude using $\llbracket \mathrm{Prf}\ \top \rrbracket^3_{\phi,\psi} = \llbracket \top \rrbracket^3_{\phi,\psi}$. The interpretations of all constructive connectives and quantifiers are described in Figure 5.

▶ **Lemma 12** (Well-typedness). *For any $i \in \{ 1, 2 \}$ and declaration $c : T$ in ($\top$-decl), ($\bot$-decl), ($\neg$-decl), ($\exists$-decl), ($\wedge$-decl), and ($\vee$-decl), we have $\llbracket c \rrbracket^{i+1}_{\phi_1,\ldots,\phi_i} \in \llbracket T \rrbracket^i_{\phi_1,\ldots,\phi_{i-1}}$.*

Proof. We check every case.

If $i = 1$: $\llbracket c \rrbracket^2_\phi = e$ and $\llbracket T \rrbracket^1 = \{ e \}$

If $i = 2$: $\llbracket \top \rrbracket^3_{\phi,\psi}$ and $\llbracket \bot \rrbracket^3_{\phi,\psi}$ are elements of $\mathcal{B}$, which is equal to $\llbracket \mathrm{Prop} \rrbracket^2_\phi$,

- $\llbracket \neg \rrbracket^3_{\phi,\psi}$ is an element of $\mathcal{F}(\{ e \} \times \mathcal{B}, \mathcal{B})$, which is equal to $\llbracket \mathrm{Prop} \to \mathrm{Prop} \rrbracket^2_\phi$,
- $\llbracket \wedge \rrbracket^3_{\phi,\psi}$ and $\llbracket \vee \rrbracket^3_{\phi,\psi}$ are elements of $\mathcal{F}(\{ e \} \times \mathcal{B}, \mathcal{F}(\{ e \} \times \mathcal{B}, \mathcal{B}))$, which is equal to $\llbracket \mathrm{Prop} \to \mathrm{Prop} \to \mathrm{Prop} \rrbracket^2_\phi$,
- and $\llbracket \exists \rrbracket^3_{\phi,\psi}$ is an element of $\mathcal{F}(\mathcal{A} \times \mathcal{B}, \mathcal{F}(\{ e \} \times \mathcal{F}(\{ e \} \times S, \mathcal{B}), \mathcal{B}))$, which is equal to $\llbracket (x : \mathrm{Set}) \to (\mathrm{El}\ x \to \mathrm{Prop}) \to \mathrm{Prop} \rrbracket^2_\phi$. ◀

▶ **Lemma 13** (Validity of the axioms). *For all declarations $c : T$ in ($\top$-decl), ($\bot$-decl), ($\neg$-decl), ($\exists$-decl), ($\wedge$-decl), and ($\vee$-decl), then $\llbracket T \rrbracket^3_{\phi,\psi} \geq \tilde{\top}$.*

Proof. There are four possibilities.

- For ($\top$-decl) and ($\bot$-decl), we have $T = \mathrm{Prop}$ and by definition $\llbracket T \rrbracket^3_{\phi,\psi} = \tilde{\top} \geq \tilde{\top}$.
- For ($\neg$-decl), we have $T = \mathrm{Prop} \to \mathrm{Prop}$, and as a consequence $\llbracket T \rrbracket^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{ \tilde{\top} \}) \geq \tilde{\top}$.
- For ($\wedge$-decl) and ($\vee$-decl), we have $T = \mathrm{Prop} \to \mathrm{Prop} \to \mathrm{Prop}$ and by definition $\llbracket T \rrbracket^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{ \tilde{\Pi}(\tilde{\top}, \{ \tilde{\top} \}) \}) \geq \tilde{\top}$.
- For ($\exists$-decl), we have $T = (x : \mathrm{Set}) \to (\mathrm{El}\ x \to \mathrm{Prop}) \to \mathrm{Prop}$, and as a consequence $\llbracket T \rrbracket^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{ \tilde{\Pi}(\tilde{\Pi}(c, \{ \tilde{\top} \}), \{ \tilde{\top} \}) \ :\ c \in \mathcal{B} \}) \geq \tilde{\top}$. ◀

$$
\begin{aligned}
\llbracket \mathtt{TYPE} \rrbracket^1 = \llbracket \mathtt{KIND} \rrbracket^1 &= \mathcal{V} \\
\llbracket \mathrm{Set} \rrbracket^1 &= \mathcal{A} \\
\llbracket (x : C) \to\ D \rrbracket^1 &= \begin{cases} \{\, e \,\} & \text{if } \llbracket D \rrbracket^1 = \{\, e \,\} \\ \mathcal{F}(\llbracket C \rrbracket^1, \llbracket D \rrbracket^1) & \text{else} \end{cases} \\
\llbracket \lambda x : C.\, t \rrbracket^1 = \llbracket t\ u \rrbracket^1 &= \llbracket t \rrbracket^1 \\
\llbracket t \rrbracket^1 &= \{\, e \,\} \text{ in every other case}
\end{aligned}
$$

(a) First level of interpretation.

$$
\begin{aligned}
\llbracket \mathtt{TYPE} \rrbracket^2_\phi = \llbracket \mathtt{KIND} \rrbracket^2_\phi = \llbracket \mathrm{Set} \rrbracket^2_\phi = \llbracket o \rrbracket^2_\phi &= \mathcal{B} \\
\llbracket \mathrm{El} \rrbracket^2_\phi &= S \mapsto S \in \mathcal{F}(\mathcal{A}, \mathcal{V}) \\
\llbracket \mathrm{Prf} \rrbracket^2_\phi &= e \mapsto \{\, e \,\} \in \mathcal{F}(\{\, e \,\}, \mathcal{V}) \\
\llbracket \iota \rrbracket^2_\phi &= \{\, e \,\} \\
\llbracket (x : C) \to\ D \rrbracket^2_\phi &= \begin{cases} \{\, e \,\} \text{ if for all } c' \in \llbracket C \rrbracket^1,\ \llbracket D \rrbracket^2_{\phi, x = c'} = \{\, e \,\} \\ \mathcal{F}_d(\llbracket C \rrbracket^1 \times \llbracket C \rrbracket^2_\phi, (\llbracket D \rrbracket^2_{\phi, x = c'})_{\langle c, c' \rangle}) \text{ else} \end{cases} \\
\llbracket \leadsto \rrbracket^2_\phi &= \begin{cases} \{\, e \,\} \text{ if } T = \{\, e \,\} \\ \langle S, T \rangle \in \mathcal{A} \times \mathcal{A} \mapsto \mathcal{F}(\{\, e \,\} \times S, T) \text{ else} \end{cases} \\
\llbracket x \rrbracket^2_\phi &= \phi(x) \\
\llbracket \lambda x : C.\, t \rrbracket^2_\phi &= \begin{cases} e \text{ if for all } c \in \llbracket C \rrbracket^1,\ \llbracket t \rrbracket^2_{\phi, x = c} = e \\ c \in \llbracket C \rrbracket^1 \mapsto \llbracket t \rrbracket^2_{\phi, x = c} \text{ else} \end{cases} \\
\llbracket t\ u \rrbracket^2_\phi &= \begin{cases} e \text{ if } \llbracket t \rrbracket^2_\phi = e \\ \llbracket t \rrbracket^2_\phi\ \llbracket u \rrbracket^2_\phi \text{ else} \end{cases} \\
\llbracket t \rrbracket^2_\phi &= e \text{ in every other case}
\end{aligned}
$$

(b) Second level of interpretation.

$$
\begin{aligned}
\llbracket \mathtt{TYPE} \rrbracket^3_{\phi,\psi} = \llbracket \mathtt{KIND} \rrbracket^3_{\phi,\psi} &= \tilde{\top} \\
\llbracket \mathrm{Set} \rrbracket^3_{\phi,\psi} = \llbracket \iota \rrbracket^3_{\phi,\psi} = \llbracket o \rrbracket^3_{\phi,\psi} &= \tilde{\top} \\
\llbracket \leadsto \rrbracket^3_{\phi,\psi} &= \langle \langle S, a \rangle, \langle T, b \rangle \rangle \in (\mathcal{A} \times \mathcal{B})^2 \mapsto \tilde{\Pi}(a, \{\, b \,\}) \in \mathcal{B} \\
\llbracket \mathrm{El} \rrbracket^3_{\phi,\psi} &= \langle S, a \rangle \in \mathcal{A} \times \mathcal{B} \mapsto a \in \mathcal{B} \\
\llbracket \mathrm{Prf} \rrbracket^3_{\phi,\psi} &= \langle e, a \rangle \in \{\, e \,\} \times \mathcal{B} \mapsto a \in \mathcal{B} \\
\llbracket \Rightarrow \rrbracket^3_{\phi,\psi} &= \langle e, a \rangle \in \{\, e \,\} \times \mathcal{B} \mapsto \langle e, b \rangle \in \{\, e \,\} \times \mathcal{B} \mapsto \tilde{\Pi}(a, \{\, b \,\}) \\
\llbracket \forall \rrbracket^3_{\phi,\psi} &= \langle S, a \rangle \in \mathcal{A} \times \mathcal{B} \mapsto \langle e, g \rangle \in \{\, e \,\} \times \mathcal{F}(\{\, e \,\} \times S, \mathcal{B}) \mapsto \\
&\quad \tilde{\Pi}(a, \{\, g\ \langle e, s \rangle \ :\ s \in S \,\}) \\
\llbracket x \rrbracket^3_{\phi,\psi} &= \psi(x) \\
\llbracket (x : C) \to\ D \rrbracket^3_{\phi,\psi} &= \tilde{\Pi} \left(\llbracket C \rrbracket^3_{\phi,\psi}, \{\, \llbracket D \rrbracket^3_{\phi(x = c'), \psi(x = c)} \ :\ c' \in \llbracket C \rrbracket^1, c \in \llbracket C \rrbracket^2_\phi \,\} \right) \\
\llbracket \lambda x : C.\, t \rrbracket^3_{\phi,\psi} &= \begin{cases} e \text{ if for all } \langle c', c \rangle \in \llbracket C \rrbracket^1 \times \llbracket C \rrbracket^2_\phi,\ \llbracket t \rrbracket^3_{\phi(x = c'), \psi(x = c)} = e \\ \langle c', c \rangle \in \llbracket C \rrbracket^1 \times \llbracket C \rrbracket^2_\phi \mapsto \llbracket t \rrbracket^3_{\phi(x = c'), \psi(x = c)} \text{ else} \end{cases} \\
\llbracket t\ u \rrbracket^3_{\phi,\psi} &= \begin{cases} e \text{ if } \llbracket t \rrbracket^3_{\phi,\psi} = e \\ \llbracket t \rrbracket^3_{\phi,\psi} \langle \llbracket u \rrbracket^2_\phi, \llbracket u \rrbracket^3_{\phi,\psi} \rangle \text{ else} \end{cases}
\end{aligned}
$$

(c) Third level of interpretation.

Figure 4 Model of minimal STT.

$$[\![t]\!]^1 = \{\, e \,\} \text{ if } t \in \{\, \top, \bot, \neg, \exists, \wedge, \vee, \mathrm{Prop} \,\}$$

(a) First level.

$$\begin{aligned} [\![\mathrm{Prop}]\!]^2_\phi &= \mathcal{B} \\ [\![t]\!]^2_\phi &= e \text{ if } t \in \{\, \top, \bot, \neg, \exists, \wedge, \vee \,\} \end{aligned}$$

(b) Second level.

$$\begin{aligned}
[\![\mathrm{Prop}]\!]^3_{\phi,\psi} &= \tilde{\top} \\
[\![\top]\!]^3_{\phi,\psi} &= \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(c, \{\, c \,\}) \;:\; c \in \mathcal{B} \,\}) \\
[\![\bot]\!]^3_{\phi,\psi} &= \tilde{\Pi}(\tilde{\top}, \mathcal{B}) \\
[\![\wedge]\!]^3_{\phi,\psi} &= \langle e, a \rangle \mapsto \langle e, b \rangle \mapsto \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(\tilde{\Pi}(a, \{\, \tilde{\Pi}(b, \{\, c \,\}) \,\}), \{\, c \,\}) \;:\; c \in \mathcal{B} \,\}) \\
[\![\vee]\!]^3_{\phi,\psi} &= \langle e, a \rangle \mapsto \langle e, b \rangle \mapsto \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(\tilde{\Pi}(a, \{\, c \,\}), \{\, \tilde{\Pi}(\tilde{\Pi}(b, \{\, c \,\}), \{\, c \,\}) \,\}) \;:\; c \in \mathcal{B} \,\}) \\
[\![\neg]\!]^3_{\phi,\psi} &= \langle e, a \rangle \mapsto \tilde{\Pi}(a, \{\, \tilde{\Pi}(\tilde{\top}, \mathcal{B}) \,\}) \\
[\![\exists]\!]^3_{\phi,\psi} &= \langle S, a \rangle \in \mathcal{A} \times \mathcal{B} \mapsto \langle e, g \rangle \in \{\, e \,\} \times \mathcal{F}(\{\, e \,\} \times S, \mathcal{B}) \mapsto \\
&\quad \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(a, \{\, \tilde{\Pi}(g\ \langle e, s \rangle, \{\, c \,\}) \;:\; s \in S \,\}), \{\, c \,\}) \;:\; c \in \mathcal{B} \,\})
\end{aligned}$$

(c) Third level.

Figure 5 Interpretations of Constructive STT connectives.

▶ **Lemma 14** (Validity of the congruence). *For all rewrite rules $\ell \hookrightarrow r$ in ($\top$-red), ($\bot$-red), ($\neg$-red), ($\exists$-red), ($\wedge$-red), and ($\vee$-red), then for all $i \in \{\, 1, 2, 3 \,\}$, $[\![\ell]\!]^i_{\phi_1,\ldots,\phi_{i-1}} = [\![r]\!]^i_{\phi_1,\ldots,\phi_{i-1}}$.*

Proof. We check every case.

If $i = 1$: in all cases $[\![\ell]\!]^1 = [\![r]\!]^1 = \{\, e \,\}$.

If $i = 2$: in all cases $[\![\ell]\!]^2_\phi = [\![r]\!]^2_\phi = e$.

If $i = 3$: we check the equality for every rule.

($\top$-red): $[\![\mathrm{Prf}\ \top]\!]^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(c, \{\, c \,\}) \;:\; c \in \mathcal{B} \,\}) = [\![(z : \mathrm{Prop}) \to \mathrm{Prf}\ z \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

($\bot$-red): $[\![\mathrm{Prf}\ \bot]\!]^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \mathcal{B}) = [\![(z : \mathrm{Prop}) \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

($\neg$-red): $[\![\mathrm{Prf}\ \neg\, A]\!]^3_{\phi,\psi} = \tilde{\Pi}([\![A]\!]^3_{\phi,\psi}, \{\, \tilde{\Pi}(\tilde{\top}, \mathcal{B}) \,\}) = [\![\mathrm{Prf}\ A \to (z : \mathrm{Prop}) \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

($\wedge$-red): $[\![\mathrm{Prf}\ A \wedge B]\!]^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(\tilde{\Pi}([\![A]\!]^3_{\phi,\psi}, \{\, \tilde{\Pi}([\![B]\!]^3_{\phi,\psi}, \{\, c \,\}) \,\}), \{\, c \,\}) \;:\; c \in \mathcal{B} \,\}) = [\![(z : \mathrm{Prop}) \to (\mathrm{Prf}\ A \to \mathrm{Prf}\ B \to \mathrm{Prf}\ z) \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

($\vee$-red): $[\![\mathrm{Prf}\ A \vee B]\!]^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}(\tilde{\Pi}([\![A]\!]^3_{\phi,\psi}, \{\, c \,\}), \{\, \tilde{\Pi}(\tilde{\Pi}([\![B]\!]^3_{\phi,\psi}, \{\, c \,\}), \{\, c \,\}) \,\}) \;:\; c \in \mathcal{B} \,\}) = [\![(z : \mathrm{Prop}) \to (\mathrm{Prf}\ A \to \mathrm{Prf}\ z) \to (\mathrm{Prf}\ B \to \mathrm{Prf}\ z) \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

($\exists$-red): $[\![\mathrm{Prf}\ (\exists\ T\ P)]\!]^3_{\phi,\psi} = \tilde{\Pi}(\tilde{\top}, \{\, \tilde{\Pi}([\![T]\!]^3_{\phi,\psi}, \{\, \tilde{\Pi}([\![P]\!]^3_{\phi,\psi}\ \langle e, s \rangle, \{c\}) \;:\; s \in [\![T]\!]^2_\phi \,\}), \{c\}) \;:\; c \in \mathcal{B} \,\}) = [\![(z : \mathrm{Prop}) \to ((y : \mathrm{El}\ T) \to \mathrm{Prf}\ P\ y \to \mathrm{Prf}\ z) \to \mathrm{Prf}\ z]\!]^3_{\phi,\psi}$

In all cases, reduction preserves interpretation. ◀

▶ **Proposition 15.** *$\hookrightarrow_\beta$ strongly terminates on well-typed terms of Constructive STT.*

Proof. Constructive STT has models valued in all full, ordered, and complete Π-algebras. Thus, the theory is super-consistent and we conclude by Theorem 9. ◀

▶ **Corollary 16.** *Ecumenical STT is weakly normalizing.*

Proof. We exhibit a normalizing strategy for a given well-typed term t of Ecumenical STT.

1. First, normalize t with respect to every rule defining classical connectives and quantifiers, *i.e.* ($\Rightarrow_c$-red), ($\wedge_c$-red), ($\vee_c$-red), ($\forall_c$-red), and ($\exists_c$-red). This procedure terminates as the number of classical connectives and quantifiers stricly decreases with every reduction. Note that the result t' of this procedure is well-typed in Constructive STT.
2. Second, β-normalize t'. This procedure terminates by Proposition 15, yielding a term t''.

3. Finally, normalize t'' with respect to ($\top$-red), ($\bot$-red), ($\Rightarrow$-red), ($\wedge$-red), ($\vee$-red), ($\forall$-red), and ($\exists$-red). The number of connectives and quantifiers strictly decreases with each reduction. Each of these reduction step using rules ($\forall$-red) and ($\exists$-red) might create one β-redex; however it is of the form $(\lambda x : t.\, u)\ \ y$, which can be immediately reduced without increasing the number of connectives and quantifiers.

The term resulting from this procedure is $\hookrightarrow_{\beta\mathcal{R}^{e}_{HO}}$-normal. ◀

We can conclude that type-checking in the logical fragments of theory $\mathcal{U}$ is decidable.

▶ **Corollary 17.** *Type-checking in Ecumenical STT is decidable.*

In practice, this normalization strategy is not implemented to type-check modulo Ecumenical STT. However, a considerable number of proofs, notably the standard HOL Light library, have been type-checked in this theory [12]. The diversity and size of these developments provide no counter-example to the strong normalization of Ecumenical STT.

▶ **Conjecture 18.** *Ecumenical STT is strongly normalizing.*

4 First order ecumenism

Some fragments of theory $\mathcal{U}$ have been previously studied separately in [3, 26, 43]. Soundness and conservativity of these expressions with respect to a reference system, and consistency have not all been established for all fragments; the consistency of theory $\mathcal{U}$ is still an open question.

In the following sections, we study the soundness and conservativity of all logical fragments of $\mathcal{U}$ with respect to appropriate reference systems (first order constructive and classical logics, and higher-order constructive and classical logics). We also establish the consistency of the logical fragments of theory $\mathcal{U}$, which is a first step towards the consistency of the whole theory. In the current section, we focus on the first order fragments, ie Constructive Predicate Logic and Ecumenical Predicate Logic.

4.1 Reference systems: constructive and classical predicate logic

As reference systems for Constructive Predicate Logic and Ecumenical Predicate Logic, we choose the systems NJ and NK [11].

In these systems, terms are defined over a first order language $\mathcal{L}$ containing *function* and *predicate* symbols with their arity. Terms are of the form $t, u, \dots = x \mid f(t_1, \dots, t_n)$ where x is a variable and $f \in \mathcal{L}$ is a function symbol of arity n. Formulas are defined by $A, B, \dots = P(t_1, \dots, t_n) \mid \top \mid \bot \mid A \wedge B \mid A \vee B \mid A \Rightarrow B \mid \neg A \mid \forall x.\ A \mid \exists x.\ A$, where x is a variable and $P \in \mathcal{L}$ is an n-ary predicate symbol. Typing contexts are defined by $\Gamma, \Delta, \dots = [] \mid \Gamma, A$. The rules of the NJ proof system are described in Figure 6. System NK is the extension of NJ with the excluded-middle rule (EM) of conclusion $A \vee \neg A$ and without premises. We respectively write $\Gamma \vdash_{\mathsf{NJ}} A$ and $\Gamma \vdash_{\mathsf{NK}} A$ if the judgment $\Gamma \vdash A$ is derivable in NJ and NK.

4.2 Soundness and conservativity of Constructive Predicate Logic

Soundness and conservativity of Constructive Predicate Logic were first proved in [13], and restated and reproven in [40]. The proof if soundness is tedious and error prone: in both aforementioned proofs, the free variables occuring in the constructive natural deduction proof are not accurately taken into account. In the following, we reprove soundness and conservativity of Constructive and highlight the errors made in previous proofs.

$$\frac{A \in \Gamma}{\Gamma \vdash A}\ \text{axiom} \qquad \frac{}{\Gamma \vdash \top}\ \top\text{-intro} \qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A}\ \bot\text{-elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A}\ \wedge\text{-elim} \qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B}\ \wedge\text{-elim}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B}\ \wedge\text{-intro} \qquad \frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A}\ \neg\text{-intro} \qquad \frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot}\ \neg\text{-elim} \qquad \frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B}\ \Rightarrow\text{-intro}$$

$$\frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B}\ \Rightarrow\text{-elim} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash A \vee B}\ \vee\text{-intro} \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B}\ \vee\text{-intro}$$

$$\frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C}\ \vee\text{-elim} \qquad \frac{\Gamma \vdash \forall x.\ A}{\Gamma \vdash A}\ \forall\text{-elim} \qquad \frac{\Gamma \vdash A}{\Gamma \vdash \exists x.\ A}\ \exists\text{-intro}$$

$$\frac{\Gamma \vdash A \quad x \notin fv(\Gamma)}{\Gamma \vdash \forall x.\ A}\ \forall\text{-intro} \qquad \frac{\Gamma \vdash \exists x.\ A \quad \Gamma, A \vdash B \quad x \notin fv(\Gamma, B)}{\Gamma \vdash B}\ \exists\text{-elim}$$

Figure 6 Rules of constructive predicate logic NJ.

$$\begin{array}{rcl} |x|_c & \triangleq & x \\ |f(t_1, \ldots, t_n)|_c & \triangleq & \dot{f}\ |t_1|_c \ldots |t_n|_c \\ |P(t_1, \ldots, t_n)|_c & \triangleq & \dot{P}\ |t_1|_c \ldots |t_n|_c \\ |\top|_c & \triangleq & \top \\ |\bot|_c & \triangleq & \bot \\ |\neg A|_c & \triangleq & \neg\,|A|_c \end{array} \qquad \begin{array}{rcl} |A \wedge B|_c & \triangleq & |A|_c \wedge |B|_c \\ |A \vee B|_c & \triangleq & |A|_c \vee |B|_c \\ |A \Rightarrow B|_c & \triangleq & |A|_c \Rightarrow |B|_c \\ |\forall x.\ A|_c & \triangleq & \forall\ \iota\ (\lambda x : \text{El}\ \iota.\,|A|_c) \\ |\exists x.\ A|_c & \triangleq & \exists\ \iota\ (\lambda x : \text{El}\ \iota.\,|A|_c) \end{array}$$

Figure 7 Translation $|\cdot|_c$ of NJ to its expression in $\lambda\Pi / \equiv$.

As in [40], a first order language $\mathcal{L}$ is encoded in $\lambda\Pi / \equiv$ by a context $\Delta_{\mathcal{L}}$ declaring an n-ary function $\dot{f} : \text{El}\ \iota \to \ldots \to \text{El}\ \iota \to \text{El}\ \iota$ for every n-ary function symbol $f \in \mathcal{L}$, and an n-ary function $\dot{P} : \text{El}\ \iota \to \ldots \to \text{El}\ \iota \to \text{Prop}$ for every n-ary predicate symbol $P \in \mathcal{L}$. For example, the first order language $\mathcal{L}$ containing a nullary predicate P and a unary predicate Q is expressed in $\lambda\Pi / \equiv$ by the context $\Delta_{\mathcal{L}}$ represented in Figure 8. Terms and formulas of the intuitionistic first order system NJ are naturally embedded into Constructive Predicate Logic using transformation $|.|_c$ defined in Figure 7. Finally, for any context $\Gamma = A_1, \ldots, A_n$ and proposition A, denoting by $y_1, \ldots, y_k$ the free variables of Γ, A, we define $|\Gamma|_c^A = y_1 : \text{El}\ \iota, \ldots, y_k : \text{El}\ \iota, x_1 : \text{Prf}\ |A_1|_c, \ldots, x_n : \text{Prf}\ |A_n|_c$. Again, Figure 8 provides an example: the NJ context $\forall x.\ [Q(x) \wedge P]$ is translated into the $\lambda\Pi / \equiv$ context Γ.

In [40], the NJ proof represented in Figure 8 is expressed by the term π in Constructive Predicate Logic. Note that the variable y is not free in the final judgment $\forall x.\ [Q(x) \wedge P] \vdash P$, thus is not declared in the typing context $\Delta_{\mathcal{L}}, \Gamma$. However, y is free in the NJ proof, thus appears free in term π. As a consequence, π is not well-typed in $\Delta_{\mathcal{L}}, \Gamma$ and the soundness result of [40] does not hold. To handle such free variables, we suggest the following slight alteration: we add a witness $w : \text{El}\ \iota$ to context $\Delta_{\mathcal{L}}$ and substitute whenever necessary. As an example, we express the NJ proof of Figure 8 with the term $\pi(y/w)$, well-typed in $\Delta_{\mathcal{L}}, \Gamma$. Note that this presentation can be easily extended to many sorted natural deduction by adding a constant $s : \text{El}$ and a witness $w_s : \text{El}\ s$ for every additional sort.

▶ **Lemma 19** (Soundness). *If A is an NJ formula over a first-order language $\mathcal{L}$ such that $\Gamma \vdash_{NJ} A$, then there is a term $t \in \Lambda(\Sigma^c_{FO})$ such that $\Delta_{\mathcal{L}}; |\Gamma|^A_c \vdash^c_{FO} t : \text{Prf}\ |A|_c$.*

$$\dfrac{\dfrac{\forall x.\ [Q(x)\wedge P] \vdash \forall x.\ [Q(x)\wedge P]}{\forall x.\ [Q(x)\wedge P] \vdash Q(y)\wedge P}\ \forall\text{-elim}}{\forall x.\ [Q(x)\wedge P]\vdash P}\ \wedge\text{-elim}$$

$$\begin{aligned}\Delta_{\mathscr{L}} &= (P:\mathsf{Prop}), (Q:\mathsf{El}\ \iota \to \mathsf{Prop})\\ \Gamma &= (H:\mathsf{Prf}\ (\forall\ \iota\ (\lambda x\!:\!\mathsf{El}\ \iota.\,[(Q\ x)\wedge P])))\\ \pi &= (H\ y)\ P\ (\lambda x_1\!:\!\mathsf{Prf}\ (Q\ y).\,\lambda x_2\!:\!\mathsf{Prf}\ P.\,x_2)\end{aligned}$$

Figure 8 Counter-example to soundness proofs of constructive predicate logic from [13, 40].

Proof. By induction on the derivation of $\Gamma \vdash A$. We develop the case of the left elimination of the conjunction $\wedge$-elim to illustrate the use of the witness $w:\mathsf{El}\ \iota$. By induction hypothesis on the proof π_B of $\Gamma \vdash A\wedge B$, there is a term t_B such that $\Delta_{\mathscr{L}}; |\Gamma|_c^{A\wedge B} \vdash_{FO}^{c} t_B : \mathsf{Prf}\ |A\wedge B|_c$.

Term $t = t_B\ |A|_c\ (\lambda z_1\!:\!\mathsf{Prf}\ |A|_c.\,\lambda z_2\!:\!\mathsf{Prf}\ |B|_c.\,z_1)$ is of type $\mathsf{Prf}\ |A|_c$ in context $\Delta_{\mathscr{L}}, |\Gamma|_c^{A\wedge B}$. Let $y_1,\dots,y_k = fv(B)\backslash fv(\Gamma, A)$. As $|\Gamma|_c^{A\wedge B} = |\Gamma|_c^{A}, y_1 : \mathsf{El}\ \iota, \dots, y_k : \mathsf{El}\ \iota$, and using the substitution lemma [40, Lemma 2.6.9.], $\Delta_{\mathscr{L}}; |\Gamma|_c^{A} \vdash (w/y_1,\dots,w/y_k)t : \mathsf{Prf}\ |A|_c$. ◀

The conservativity of Constructive Predicate Logic with respect to NJ, which is the converse statement to Lemma 19, has been established in [13, 40] and can be seen as a specific case of the proof of conservativity of Constructive STT, further developed in Section 5.2.2.

▶ **Lemma 20** (Conservativity). *Let $\mathscr{L}$ be a first-order language and A a NK formula over $\mathscr{L}$. If there is a term $t \in \Lambda(\Sigma_{FO}^{c})$ such that $\Delta_{\mathscr{L}}; |\Gamma|_c^{A} \vdash_{FO}^{c} t : \mathsf{Prf}\ |A|_c$, then $\Gamma \vdash_{NJ} A$.*

4.3 Soundness and conservativity of Ecumenical Predicate Logic

In the following section, the soudness and conservativity of Ecumenical Predicate Logic with respect to NK are established using the analogous results already established for Constructive Predicate Logic, *i.e.* Lemmas 19 and 20, and the properties of double-negation translations.

Figure 9a defines the embedding $|.|_e$ of NK formulas into Ecumenical terms, which maps every NK connective to the corresponding classical connective. Considering the construction of the classical connectives, this transformation mimicks the Kolmogorov double negation translation $A \mapsto \neg\neg(A^{\perp})$ [29] represented in Figure 9b. In the following, the Kolmogorov translation $\neg\neg(\cdot^{\perp})$ is naturally extended to contexts.

$$\begin{array}{rcl} |x|_e &=& x\\ |f(t_1,\dots,t_n)|_e &=& \dot{f}\ |t_1|_e \dots |t_n|_e\\ |P(t_1,\dots,t_n)|_e &=& \dot{P}\ |t_1|_e \dots |t_n|_e\\ |\square|_e &=& \square\\ |\neg A|_e &=& \neg\,|A|_e\\ |A \bowtie B|_e &=& |A|_e \bowtie_c |B|_e\\ |Qx.\ A|_e &=& Q_c\ \iota\ (\lambda x\!:\!\mathsf{El}\ \iota.\,|A|_e)\end{array}$$

(a) Expressing NK into Ecumenical Predicate Logic.

$$\begin{array}{rcl} P(t_1,\dots,t_n)^{\perp} &=& P(t_1,\dots,t_n)\\ \square^{\perp} &=& \square\\ (\neg A)^{\perp} &=& \neg A\\ (A \bowtie B)^{\perp} &=& (\neg\neg A) \bowtie (\neg\neg B)\\ (Qx.\ A)^{\perp} &=& Qx.\ (\neg\neg A)\end{array}$$

(b) The $\cdot^{\perp}$ translation from NK to NJ.

Figure 9 Translations of NK propositions, where $\square \in \{\top, \bot\}$, $\bowtie \in \{\wedge, \vee, \Rightarrow\}$, and $Q \in \{\forall, \exists\}$.

▶ **Lemma 21.** *For every NK proposition A, $\mathsf{Prf}_c\ \ |A|_e \equiv_{\beta\mathscr{R}_{FO}^{e}} |\neg\neg A^{\perp}|_c$.*

Proof. By a straightforward induction on the structure of formula A. ◀

▶ **Lemma 22** ([29]). *For every* NK *proposition A, if $\Gamma \vdash_{\mathsf{NK}} A$ then $\neg\neg\Gamma^{\perp} \vdash_{\mathsf{NJ}} \neg\neg A^{\perp}$.*

The soundness of Ecumenical Predicate Logic with respect to NK is immediate using Lemmas 21 and 22. Formally, for any context $\Gamma = A_1, \dots, A_n$ and proposition A, denoting by $y_1, \dots, y_k$ the free variables of Γ, A, we define $|\Gamma|_e^A = y_1 : \mathrm{El}\ \iota, \dots, y_k : \mathrm{El}\ \iota, x_1 : \mathrm{Prf}_c\ |A_1|_e, \dots, x_n : \mathrm{Prf}_c\ |A_n|_e$.

▶ **Lemma 23** (Soundness). *If A is* NK *formula over language $\mathcal{L}$ such that $\Gamma \vdash_{\mathsf{NK}} A$ is provable in* NK*, then there exists a term $t \in \Lambda(\Sigma_{FO}^e)$ such that $\Delta_{\mathcal{L}}; |\Gamma|_{A,c} \vdash_{FO}^e t : \mathit{Prf}_c\ |A|_e$.*

Proof. By Lemma 22, $\neg\neg\Gamma^{\perp} \vdash_{\mathsf{NJ}} \neg\neg A^{\perp}$ is provable in NJ. By the soundness of Constructive Predicate Logic, $\Delta_{\mathcal{L}}; |\neg\neg\Gamma^{\perp}|_c^{\neg\neg A} \vdash_{FO}^c t : \mathrm{Prf}\ |\neg\neg A^{\perp}|_c$. By Lemma 21 and the fact that $fv(\neg\neg A) = fv(A)$, we conclude that $\Delta_{\mathcal{L}}; |\Gamma|_e^A \vdash_{FO}^e t : \mathrm{Prf}_c\ |A|_e$. ◀

▶ **Lemma 24** (Conservativity). *Let $\mathcal{L}$ be a first-order language and A a* NK *formula over $\mathcal{L}$. If there is a term $t \in \Lambda(\Sigma_{FO}^e)$ such that $\Delta_{\mathcal{L}}, |\Gamma|_e^A \vdash_{FO}^e t : \mathit{Prf}_c\ |A|_e$, then $\Gamma \vdash_{\mathsf{NK}} A$.*

Proof. By the conversion rule and Lemma 21, $\Delta_{\mathcal{L}}, |\neg\neg\Gamma^{\perp}|_c^A \vdash_{FO}^e t : \mathrm{Prf}\ (|\neg\neg A^{\perp}|_c)$. By Corollary 3, there is a reduct of $\mathrm{Prf}\ (|\neg\neg A^{\perp}|_c)$, which we will denote by T, such that $\Delta_{\mathcal{L}}, |\neg\neg\Gamma^{\perp}|_c^A \vdash_{FO}^c t : T$. By conversion, $\Delta_{\mathcal{L}}, |\neg\neg\Gamma^{\perp}|_c^A \vdash_{FO}^c t : \mathrm{Prf}\ (|\neg\neg A^{\perp}|_c)$. By Lemma 20, $\neg\neg\Gamma^{\perp} \vdash_{\mathsf{NJ}} \neg\neg A^{\perp}$ is derivable in NJ, and we conclude by Lemma 22 that $\Gamma \vdash_{\mathsf{NK}} A$. ◀

5 Higher order ecumenism

In the following section, we study the soundness and conservativity of the higher order logical fragments of $\mathcal{U}$ with respect to higher-order constructive and classical logics. We will then conclude that Ecumenical STT is consistent.

5.1 Reference systems: constructive and classical HOL-λ

As reference systems for Constructive and Ecumenical STT, we consider the intentional version of HOL-λ [16], that is the system obtained by removing the η-expansion rule.

The types of the system HOL-λ are the *simple types* defined by $T, U, \dots = \iota \mid o \mid T \to U$. Terms of HOL-$\lambda$ are λ-terms with additional constants, among which figure the logical connectives. Formally, terms and their associated types are inductively defined by:

- a set of typed variables $\mathcal{X}$, such that every variable $x \in \mathcal{X}$ of type T is a term of type T;
- a set of typed constants $\mathcal{L}$, such that every constant $c \in \mathcal{L}$ of type T is a term of type T;
- for every term t of type U and variable $x \in \mathcal{X}$ of type T, $\lambda x.\ t$ is a term of type $T \to U$;
- for every pair of terms t and u of respective types $T \to U$ and T, $t\ u$ is a term of type U;
- $\dot{\Rightarrow}$, $\dot{\wedge}$, and $\dot{\vee}$ are terms of type $o \to o \to o$, $\dot{\perp}$ and $\dot{\top}$ of type o, and $\dot{\neg}$ of type $o \to o$;
- $\dot{\forall}_T$ and $\dot{\exists}_T$ are terms of type $(T \to o) \to o$ for every simple type T.

We assume that there is an infinite number of variables associated to each simple type. Terms of type o are called *propositions*. Note that we will use the infix notation for the binary connectives $\dot{\Rightarrow}$, $\dot{\wedge}$, and $\dot{\vee}$ for readability purposes. We write the substitution of variable x by a similarly typed term u in a term t by $(u/x)t$.

The β-reduction is defined by the rewrite rule $(\lambda x.\ t)\ u \hookrightarrow (u/x)t$. The rewriting system $\hookrightarrow_\beta$, *i.e.* the smallest relation containing β-reduction and closed by term constructors and substitution, is confluent and strongly normalizing [23]. In the following, the $\beta-$normal form of a term t will be denoted by $t{\downarrow}$. The proof system of HOL-λ is shown on Figure 10; all propositions appearing in a proof are normal. The constructive subsystem HOL-λI of HOL-λ is the system obtained by removing the excluded-middle rule (EM) from its proof system.

$$
\dfrac{}{\Gamma \vdash A}\ \text{axiom} \qquad
\dfrac{}{\Gamma \vdash A \mathbin{\dot\vee} \dot\neg A}\ \text{EM} \qquad
\dfrac{}{\Gamma \vdash \dot\top}\ \dot\top\text{-intro} \qquad
\dfrac{\Gamma \vdash \dot\bot}{\Gamma \vdash A}\ \dot\bot\text{-elim} \qquad
\dfrac{\Gamma \vdash A \mathbin{\dot\wedge} B}{\Gamma \vdash A}\ \dot\wedge\text{-elim}
$$

$$
\dfrac{\Gamma \vdash A \mathbin{\dot\wedge} B}{\Gamma \vdash B}\ \dot\wedge\text{-elim} \qquad
\dfrac{\Gamma \vdash B \qquad \Gamma \vdash A}{\Gamma \vdash A \mathbin{\dot\wedge} B}\ \dot\wedge\text{-intro} \qquad
\dfrac{\Gamma \vdash A}{\Gamma \vdash A \mathbin{\dot\vee} B}\ \dot\vee\text{-intro} \qquad
\dfrac{\Gamma \vdash B}{\Gamma \vdash A \mathbin{\dot\vee} B}\ \dot\vee\text{-intro}
$$

$$
\dfrac{\Gamma, A \vdash B}{\Gamma \vdash A \mathbin{\dot\Rightarrow} B}\ \dot\Rightarrow\text{-intro} \qquad
\dfrac{\Gamma \vdash A \mathbin{\dot\Rightarrow} B \qquad \Gamma \vdash A}{\Gamma \vdash B}\ \dot\Rightarrow\text{-elim} \qquad
\dfrac{\Gamma \vdash (A\ t){\downarrow}}{\Gamma \vdash \dot\exists_T A}\ \dot\exists\text{-intro} \qquad
\dfrac{\Gamma \vdash \dot\forall_T A}{\Gamma \vdash (A\ t){\downarrow}}\ \dot\forall\text{-elim}
$$

$$
\dfrac{\Gamma \vdash A \mathbin{\dot\vee} B \qquad \Gamma, A \vdash C \qquad \Gamma, B \vdash C}{\Gamma \vdash C}\ \dot\vee\text{-elim} \qquad
\dfrac{\Gamma, A \vdash \dot\bot}{\Gamma \vdash \dot\neg A}\ \dot\neg\text{-intro} \qquad
\dfrac{\Gamma \vdash \dot\neg A \qquad \Gamma \vdash A}{\Gamma \vdash \dot\bot}\ \dot\neg\text{-elim}
$$

$$
\dfrac{\Gamma \vdash (A\ x){\downarrow} \qquad x \notin fv(\Gamma)}{\Gamma \vdash \dot\forall_T A}\ \dot\forall\text{-intro} \qquad
\dfrac{\Gamma \vdash \dot\exists_T A \qquad \Gamma, (A\ x) \downarrow \vdash B \qquad x \notin fv(\Gamma, B)}{\Gamma \vdash B}\ \dot\exists\text{-elim}
$$

Figure 10 Rules of the HOL-λ proof system.

$$
\begin{array}{rclcrclcrcl}
|\iota| &=& \iota & \qquad & |x|_c &=& x & \qquad & |\dot C|_c &=& C \\
|o| &=& o & & |t\ u|_c &=& |t|_c\ |u|_c & & |c|_c &=& \dot c \\
|T \to U| &=& |T| \leadsto |U| & & |\lambda x.\ t|_c &=& \lambda x : \widetilde{T}.\ |t|_c & & |\dot Q_T|_c &=& Q
\end{array}
$$

Figure 11 Shallow embedding of HOL-λI types and terms in the expression of constructive STT, where $C \in \{\Rightarrow, \wedge, \vee, \top, \bot, \neg\}$, $c \in \mathscr{L}$, $Q \in \{\forall, \exists\}$, and x is a HOL-λ variable of type T.

5.2 Soundness and conservativity of Constructive STT

Let us establish the direct correspondance between Constructive STT and HOL-λI.

5.2.1 Soundness of Constructive STT

We choose a shallow expression of HOL-λI in Constructive STT via a translation $|.|_c$ represented on Figure 11. This translation preserves λ-abstractions, applications, and β-conversion. We express simple types in $\lambda\Pi / \equiv$ using the translation $|\cdot|$ shown in Figure 11. We denote by $\widetilde{T}$ the normal form of El $|T|$, where T is a simple type. This notation allows to use normalized type anotations in order to map normal HOL-λI terms to normal terms of Constructive STT. Finally, we define a context $\Delta_{\mathscr{L}}$ declaring a symbol $\dot c : \widetilde{T}$ for every constant $c \in \mathscr{L}$ of type T and a witness w : El ι.

▶ **Lemma 25** (Preservation of $\hookrightarrow_\beta$). *If $t \hookrightarrow_\beta t'$ in HOL-λI, then $|t|_c \hookrightarrow_\beta |t'|_c$ in $\lambda\Pi / \equiv$.*

Proof. By the shallowness of translation $|\cdot|_c$. ◀

▶ **Corollary 26** (Preservation of β-conversion). *If t and t' are two convertible HOL-λI terms, then $|t|_c \equiv_{\beta\mathscr{R}^c_{HO}} |t'|_c$.*

▶ **Lemma 27** (Preservation of normality). *If t is a HOL-λI term, then t is β-normal if and only if $|t|_c$ is normal in Constructive STT.*

Proof. The direct implication is immediate by using Lemma 25. The converse statement is established by a straightforward induction on t. ◀

▶ **Lemma 28** (Preservation of types). *Let t be a HOL-λI term of type T, and $x_1, \dots, x_n$ its free variables of respective types $T_1, \dots, T_n$. We can type $|t|_c$ with: $\Delta_{\mathscr{L}}, x_1 : \mathrm{El}\ |T_1|, \dots, x_n : \mathrm{El}\ |T_n| \vdash^c_{HO} |t|_c : \mathrm{El}\ |T|$.*

Proof. By induction over the proof of typability of term t. ◀

Lemma 28 notably entails that for any HOL-λI proposition A, the $\lambda\Pi/\equiv$ term $\mathrm{Prf}\ |A|_c$ is well-typed in any context declaring the free variables of A. Note that using the witness $w : \mathrm{El}\ \iota$, every type $|T|$, where T is a HOL-λI simple type, has an element $w(T)$ in Constructive STT:

- if $T = \iota$, then w has type $\mathrm{El}\ \iota$.
- if $T = o$, then $\top$ has type $\mathrm{El}\ o \equiv_{\beta\mathcal{R}^c_{HO}} \mathrm{Prop}$.
- if $T = T_1 \to T_2$, then $\lambda x : \mathrm{El}\ |T_1|.\, w(T_2)$ has type $\mathrm{El}\ |T| \equiv_{\beta\mathcal{R}^c_{HO}} \mathrm{El}\ |T_1| \to \mathrm{El}\ |T_2|$.

We define a transformation from HOL-λI contexts into $\lambda\Pi/\equiv$ contexts: if $\Gamma = A_1, \dots, A_n$ is a HOL-λI context and A is a HOL-λI proposition, denoting by $y_1, \dots, y_k$ the free variables of Γ and A, of respective types $T_1, \dots, T_k$, then $|\Gamma|^A_c = y_1 : \mathrm{El}\ |T_1|, \dots, y_k : \mathrm{El}\ |T_k|, x_1 : \mathrm{Prf}\ |A_n|_c, \dots, x_n : \mathrm{Prf}\ |A_n|_c$. Observe that contrary to the case of predicate logic, some of the free variables may be of type $\mathrm{El}\ o \equiv_{\beta\mathcal{R}^c_{HO}} \mathrm{Prop}$.

▶ **Lemma 29** (Soundness). *If $\Gamma \vdash A$ is provable in HOL-λI, then there is a term $t \in \Lambda(\Sigma^c_{HO})$ such that $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} t : \mathrm{Prf}\ |A|_c$.*

Proof. By induction on the derivation of $\Gamma \vdash A$. We develop the cases of the introduction and elimination of the universal quantifier.

Rule $\dot\forall$-elim: By induction hypothesis, there is t_A such that $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} t_A : \mathrm{Prf}\ |\dot\forall_T A|_c$. By weakening and the application rule, $\Delta_{\mathscr{L}}, |\Gamma|^{(A\ t)}_c \vdash^c_{HO} (t_A\ t) : \mathrm{Prf}\ |A\ t|_c$. By Corollary 26 and the conversion rule, $\Delta_{\mathscr{L}}, |\Gamma|^{(A\ t)}_c \vdash^c_{HO} t_A : \mathrm{Prf}\ |(A\ t){\downarrow}\,|_c$. By the substitution lemma [40, Lemma 2.6.9.] applied to every variable of $fv(A\,t)\backslash fv(\Gamma, (A\,t){\downarrow})$ and witnesses of the associated simples types, $\Delta_{\mathscr{L}}, |\Gamma|^{(A\ t){\downarrow}}_c \vdash^c_{HO} (t_A\ t) : \mathrm{Prf}\ |(A\ t){\downarrow}\,|_c$.

Rule $\dot\forall$-intro: By induction hypothesis, there is a term t_A such that $\Delta_{\mathscr{L}}, |\Gamma|^{(A\ x){\downarrow}}_c \vdash^c_{HO} t_A : \mathrm{Prf}\ |(A\ x){\downarrow}\,|_c$. By weakening and the conversion rule, $\Delta_{\mathscr{L}}, |\Gamma|^{(A\ x)}_c \vdash^c_{HO} t_A : \mathrm{Prf}\ |A\ x|_c$. As $x \notin fv(\Gamma)$, the variable x is not declared in the context $|\Gamma|^A_c$ and we can apply the abstraction rule to the typing judgment $\Delta_{\mathscr{L}}, |\Gamma|^{A\ x}_c \vdash^c_{HO} t_A : \mathrm{Prf}\ |A\ x|_c$ to obtain a derivation of $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} \lambda x : \mathrm{El}\ |T|.\, t_A : \mathrm{Prf}\ (\forall\ |T|\ |A|_c)$. ◀

.

5.2.2 Conservativity of Constructive STT

The conservativity proof developed in this section is similar to other proofs of conservativity of higher order logics expressed in $\lambda\Pi/\equiv$ [10, 2]. Notably, we heavily rely on the normalization of Ecumenical STT; we consider normal forms to constrain the form of $\lambda\Pi/\equiv$ terms to establish the preliminary Lemmas 30–35.

▶ **Lemma 30** (Normal proof types). *If A is a HOL-λI proposition, then $(\mathrm{Prf}\ |A|_c){\downarrow}$ has the form $(x_1 : M_1) \to \dots \to (x_n : M_n) \to \mathrm{Prf}\ M$ for some terms $M, M_1, \dots, M_n$.*

▶ **Lemma 31** (Normal simple types). *If T is a simple type, the normal form of the term $\widetilde{T}$ is in the set inductively defined by $\mathrm{Prop} \mid \mathrm{El}\ s \mid \widetilde{T_1} \to \widetilde{T_2}$, where $s : \mathrm{Set}$ and $s \not\equiv_{\beta\mathcal{R}^c_{HO}} o$.*

▶ **Lemma 32** (Conserving hypotheses). *Let A and B be HOL-λI propositions and x a variable or constant. If $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} x : (x_1 : T_1) \to \dots \to (x_n : T_n) \to \mathrm{Prf}\ |B|_c$, then there is $C \in \Gamma$ such that $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} x : \mathrm{Prf}\ |C|_c$.*

▶ **Lemma 33** (Conserving simple types). *Let A be a HOL-λI proposition and $u \in \Lambda(\Sigma^c_{HO})$ a normal term. If $\Delta_{\mathscr{L}}, |\Gamma|^A_c \vdash^c_{HO} u : \mathrm{Set}$, then there is a simple type T such that $u = |T|$.*

▶ **Lemma 34** (Conserving objects). *Let A be a HOL-λI proposition, T simple type, and u a normal term in Constructive STT. If $\Delta_{\mathscr{L}}, |\Gamma|_c^A \vdash^c_{HO} u : \mathsf{El}\ |T|$, then there is a normal HOL-λI term v of type T such that $u \equiv_{\beta\mathcal{R}^c_{HO}} |v|_c$.*

▶ **Lemma 35** (Weak conservativity). *Let A be a normal HOL-λI proposition. If there is a normal term t such that $|\Gamma|_c^A \vdash^c_{HO} t : \mathsf{Prf}\ |A|_c$, then $\Gamma \vdash A$ is provable in HOL-λI.*

Proof. By induction on term t. As t is normal, there are only two cases to consider.

If $u = x\ u_1 \dots u_n$, we prove by induction on k that for any $k \in \{0, \dots, n\}$, there is a normal HOL-λI proposition A_k such that $|\Gamma|_c^A \vdash^c_{HO} x\ u_1 \dots u_k : \mathsf{Prf}\ |A_k|_c$ and $\Gamma \vdash A_k$ in HOL-λI.

If $k = 0$, then there are terms $M_1, \dots, M_n$ such that $\Delta_{\mathscr{L}}, |\Gamma|_c^A \vdash^c_{HO} x : (x_1 : M_1) \to \dots \to (x_1 : M_n) \to \mathsf{Prf}\ |A|_c$. By Lemma 32, there is $C \in \Gamma$ such that $x : \mathsf{Prf}\ |C|_c$, and as a consequence $\Delta_{\mathscr{L}}, |\Gamma|_c^A \vdash^c_{HO} x : \mathsf{Prf}\ |C|_c$ and $\Gamma \vdash C$ in HOL-λI by the axiom rule.

If $0 < k \leq n$, by induction hypothesis there is a HOL-λI proposition A_{k-1} such that $x\ u_1 \dots u_{k-1}$ has type $\mathsf{Prf}\ |A_{k-1}|_c$ in context $\Delta_{\mathscr{L}}, |\Gamma|_c^A$ and $\Gamma \vdash A_{k-1}$ in HOL-λI. By inversion, there are M_k and N_k such that $\mathsf{Prf}\ |A_{k-1}|_c \equiv_{\beta\mathcal{R}^c_{HO}} (y : M_k) \to N_k$ and u_k has type M_k in context $\Delta_{\mathscr{L}}, |\Gamma|_c^A$.
Let $\widetilde{A}_{k-1}$ be the $\beta\mathcal{R}^c_{HO}$-normal form of $|A_{k-1}|_c$. Given the form of $\mathsf{Prf}\ \widetilde{A}_{k-1}$, we know that $\widetilde{A}_{k-1}$ has a head connective; as $|\cdot|_c$ is shallow, A_{k-1} also has a head connective. We proceed by case disjunction on the head connective of A_{k-1}. Here, we develop the case of the universal quantifier, *i.e.* $A_{k-1} = \dot\forall_T B_k$ and $\mathsf{Prf}\ |A_{k-1}|_c \equiv_{\beta\mathcal{R}^c_{HO}} (y : \mathsf{El}\ |T|) \to \mathsf{Prf}\ (|B_k|_c\ y)$. By product compatibility [40, Definition 2.4.5], $M_k \equiv_{\beta\mathcal{R}^c_{HO}} \mathsf{El}\ |T|$ and $N_k \equiv_{\beta\mathcal{R}^c_{HO}} \mathsf{Prf}\ (|B_k|_c\ y)$. By Lemma 34, there is a HOL-λI term v_k of type T such that $u_k \equiv_{\beta\mathcal{R}^c_{HO}} |v_k|_c$. Using Corollary 26 and conversion, $x\ u_1 \dots u_k$ has type $\mathsf{Prf}\ (|(B_k\ v_k)\!\downarrow|_c)$ in context $\Delta_{\mathscr{L}}, |\Gamma|_c^A$. Finally, $(B_k\ v_k)\!\downarrow$ is provable in context Γ by an application of $\dot\forall$-elim.

By induction, there is a normal HOL-λI proposition A_n such that $\Delta_{\mathscr{L}}, |\Gamma|_c^A \vdash^c_{HO} t : \mathsf{Prf}\ |A_n|_c$ and $\Gamma \vdash A$. By the uniqueness of types [40, Theorem 2.6.25.], $|A_n|_c \equiv_{\beta\mathcal{R}^c_{HO}} |A|_c$. However, by Lemma 27, $|A|_c$ and $|A_n|_c$ are normal. By confluence, $|A_n|_c = |A|_c$. The embedding $|.|_c$ is injective, so $A_n = A$ and we can conclude that $\Gamma \vdash A$ in HOL-λI.

If $u = \lambda x : M.\, u_0$, then $\mathsf{Prf}\ |A|_c$ is convertible to a product $(x : M_1) \to M_2$. Using the same reasoning as in the previous case, we deduce that proposition A has a head connective and once again develop the case of the universal quantification, *i.e.* $A = \dot\forall_T B$ and $\mathsf{Prf}\ |A|_c \equiv_{\beta\mathcal{R}^c_{HO}} (x : \mathsf{El}\ |T|) \to \mathsf{Prf}\ (|B|_c\ y)$. By product compatibility [40, Definition 2.4.5], $M_1 \equiv_{\beta\mathcal{R}^c_{HO}} \mathsf{El}\ |T|$ and $M_2 \equiv_{\beta\mathcal{R}^c_{HO}} \mathsf{Prf}\ (|B|_c\ x)$. By inversion, $\Delta_{\mathscr{L}}, |\Gamma|_c^A, x : \mathsf{El}\ |T| \vdash^c_{HO} u_0 : \mathsf{Prf}\ (|B|_c\ x)$ with $x \notin fv(\Gamma, A)$. By Corollary 26, conversion, and substitution, $\Delta_{\mathscr{L}}, |\Gamma|_c^{(B\ x)\downarrow} \vdash^c_{HO} u_0 : \mathsf{Prf}\ (|(B\ x) \downarrow|_c$. By induction hypothesis, $\Gamma \vdash (B\ x)\!\downarrow$. Finally, A is provable in context Γ using $\dot\forall$-intro. ◀

Using the weak normalization of Constructive STT we finally establish conservativity.

▶ **Corollary 36** (Conservativity). *Let A be a normal HOL-λI proposition. If there is a term t such that $\Delta_{\mathscr{L}}, |\Gamma|_c^A \vdash^c_{HO} t : \mathsf{Prf}\ |A|_c$, then $\Gamma \vdash A$ is provable in HOL-λI.*

5.3 Soundness and conservativity of Ecumenical STT

Here, we show that the Ecumenical STT has a similar expressivity to the HOL-λ system. Similarly to the first order case, we use the soundness and conservativity of Constructive STT with respect to HOL-λI and the properties of the translations by double negation.

$$\begin{array}{r} |\dot{\bowtie}|_e = \bowtie_c \\ |\dot{Q}|_e = Q_c\ |T|_c \\ |t|_e = |t|_c \text{ else} \end{array} \qquad \begin{array}{rcl} x^{\perp} & = & x \\ (t\ u)^{\perp} & = & t^{\perp}\ u^{\perp} \\ (\lambda x.\ t)^{\perp} & = & \lambda x.\ t^{\perp} \end{array} \qquad \begin{array}{rcl} \dot{\bowtie}^{\perp} & = & \lambda x_1.\ \lambda x_2.\ (\dot{\neg}\dot{\neg}x_1)\dot{\bowtie}(\dot{\neg}\dot{\neg}x_2) \\ \dot{Q}^{\perp} & = & \lambda f.\ \dot{Q}(\lambda y.\ \dot{\neg}\dot{\neg}(f\ y)) \\ t^{\perp} & = & t \text{ in all other cases} \end{array}$$

(a) Defining $|\cdot|_e$. **(b)** Higher order $\perp$-translation.

Figure 12 Translations of HOL-λ terms where $\bowtie \in \{\wedge, \vee, \Rightarrow\}$, $Q \in \{\forall, \exists\}$, and x_1, x_2, f, and y are HOL-λ variables of respective types o, o, $T \to o$, and T.

Formally, we define the embedding $|.|_e$ inductively over HOL-λ terms as shown in Figure 12a. Similarly to the constructive case, transformation $|.|_e$ preserves convertibility and types, and is extended to HOL-λ contexts. However, this transformation does not map normal terms to normal terms: for example $(z\dot{\wedge}z)^{\perp} = [\lambda x_1.\ \lambda x_2.\ (\dot{\neg}\dot{\neg}x_1)\dot{\wedge}(\dot{\neg}\dot{\neg}x_2)]\ z\ z$ which is a β-redex. The proof requires a straightforward extension of the double negation translation $.^{\perp}$, represented on Figure 12b and established by Lemma 38.

▶ **Lemma 37.** *If A is convertible to B in HOL-λ, then $A^{\perp}$ is convertible to $B^{\perp}$ in HOL-λ.*

Proof. The only rule in HOL-λ is β-reduction and transformation $.^{\perp}$ acts as a morphism over abstractions and applications. ◀

▶ **Lemma 38.** *$\Gamma \vdash A$ in HOL-λ if and only if $\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow \vdash \dot{\neg}\dot{\neg}A^{\perp}\!\downarrow$ in HOL-λI.*

Proof. The forward implication is proven by induction on the derivation of $\Gamma \vdash A$. The backwards implication holds immediately, as every normal proposition A is provably equivalent to $\dot{\neg}\dot{\neg}A$ in HOL-λ, and HOL-λ is an extension of HOL-λI. ◀

▶ **Lemma 39** (Soundness)**.** *If $\Gamma \vdash A$ is provable in HOL-λI, then there is a term t such that $|\Gamma|_e^A \vdash_{HO}^e t : Prf_c\ |A|_e$ is derivable.*

Proof. By Lemmas 29 and 38, $\dot{\neg}\dot{\neg}\Gamma^{\perp}\downarrow \vdash \dot{\neg}\dot{\neg}A^{\perp}\downarrow$ in HOL-λI and there is t such that $\Delta_{\mathscr{L}}, |\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow|_c^{\dot{\neg}\dot{\neg}A^{\perp}\downarrow} \vdash_{HO}^e t : \mathrm{Prf}\ |\dot{\neg}\dot{\neg}A^{\perp}\!\downarrow|_c$. By conversion, $\Delta_{\mathscr{L}}, |\Gamma|_e^A \vdash_{HO}^e t : \mathrm{Prf}_c\ |A|_e$. ◀

▶ **Lemma 40** (Conservativity)**.** *If there is a term t such that $\Delta_{\mathscr{L}}, |\Gamma|_e^A \vdash_{HO}^e t : Prf_c\ |A|_e$, then $\Gamma \vdash A$ is provable in HOL-λ.*

Proof. If there is a term t such that $|\Gamma|_e^A \vdash_{HO}^e t : \mathrm{Prf}_c\ |A|_e$ where A is a normal HOL-λ formula, then by conversion and weakening, $|\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow|_c^{\dot{\neg}\dot{\neg}A^{\perp}\downarrow} \vdash_{HO}^e t : \mathrm{Prf}\ |\dot{\neg}\dot{\neg}A^{\perp}\!\downarrow|_c$. As t and $|\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow|_c^{\dot{\neg}\dot{\neg}A^{\perp}\downarrow}$ are respectively a term and a typing context of Constructive STT, we conclude by Corollary 3 that $|\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow|_c^{\dot{\neg}\dot{\neg}A^{\perp}\downarrow} \vdash_{HO}^e t : \mathrm{Prf}\ |\dot{\neg}\dot{\neg}A^{\perp}\!\downarrow|_c$. By conservativity of Constructive STT, $\dot{\neg}\dot{\neg}\Gamma^{\perp}\!\downarrow \vdash \dot{\neg}\dot{\neg}A^{\perp}\!\downarrow$ in HOL-λI. By Lemma 38, $\Gamma \vdash A$ in HOL-λ. ◀

▶ **Corollary 41** (Consistency)**.** *There is no derivation of $\vdash_{HO}^e \mathrm{Prf}\ \perp$ in Ecumenical STT.*

6 Conclusion

In this paper, we have studied the logical fragments of theory $\mathcal{U}$ and established their normalization, consistency, decidability of type-checking, soundness and conservativity. These results comfort the enterprise of using theory $\mathcal{U}$ to store, recheck, translate, and hybridize proofs from various proof assistants. The extension of these results, notably normalization and consistency, to the entirety of theory $\mathcal{U}$ is still an open question.

Another open question is the behaviour of hybrid propositions and proofs in Ecumenical STT, *i.e.* propositions and proofs mixing constructive and classical connectives. Given the distinct design choices made in Ecumenical STT and preexisting ecumenical systems, hybrid objects may not behave similarly. Some results over hybrid objects can however already be deduced by normalization and properties of Constructive STT, as the normal forms of hybrid objects are in this fragment.

Finally, the implementation of constructivization algorithms in theory $\mathcal{U}$ could further improve the interoperability between classical and constructive proofs. A first candidate would be the standard library of HOL Light, already translated in theory $\mathcal{U}$ [12], which could be partially exported to constructive proof assistants such as Coq, Agda, or Matita.

References

1 Claus Appel, Vincent van Oostrom, and Jakob Grue Simonsen. Higher-order (non-)modularity. In Christopher Lynch, editor, *Proceedings of the 21st International Conference on Rewriting Techniques and Applications, RTA 2010, July 11-13, 2010, Edinburgh, Scottland, UK*, volume 6 of *LIPIcs*, pages 17–32. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2010. doi:10.4230/LIPIcs.RTA.2010.17.

2 Ali Assaf. *A framework for defining computational higher-order logics. (Un cadre de définition de logiques calculatoires d'ordre supérieur)*. PhD thesis, École Polytechnique, Palaiseau, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01235303.

3 Ali Assaf, Guillaume Burel, Raphaël Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Dedukti : a Logical Framework based on the $\lambda\Pi$-Calculus Modulo Theory. Manuscript, 2016.

4 Frédéric Blanqui. *Théorie des types et réécriture. (Type theory and rewriting)*. PhD thesis, University of Paris-Sud, Orsay, France, 2001. URL: https://tel.archives-ouvertes.fr/tel-00105522.

5 Frédéric Blanqui. Size-based termination of higher-order rewriting. *Journal of Functional Programming*, 28:e11, 2018. doi:10.1017/S0956796818000072.

6 Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.FSCD.2021.20.

7 Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In Herman Geuvers, editor, *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, volume 131 of *LIPIcs*, pages 9:1–9:21. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2019. doi:10.4230/LIPIcs.FSCD.2019.9.

8 Raphaël Cauderlier and Catherine Dubois. ML pattern-matching, recursion, and rewriting: From focalize to dedukti. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing – ICTAC 2016 – 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 459–468, 2016. doi:10.1007/978-3-319-46750-4_26.

9 Raphaël Cauderlier and Catherine Dubois. Focalize and dedukti to the rescue for proof interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *Interactive Theorem Proving – 8th International Conference, ITP 2017, Brasília, Brazil, September 26-29, 2017, Proceedings*, volume 10499 of *Lecture Notes in Computer Science*, pages 131–147. Springer, 2017. doi:10.1007/978-3-319-66107-0_9.

10 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications, 8th International Conference, TLCA 2007, Paris, France, June 26-28, 2007, Proceedings*, volume 4583 of *Lecture Notes in Computer Science*, pages 102–117. Springer, 2007. doi:10.1007/978-3-540-73228-0_9.

11 Dag Prawitz. *Natural deduction, a proof-theoretical study.* PhD thesis, Stockolm: Almqvist & Wicksell, 1965.

12 Deducteam. Nubo, repository of interoperable formal proofs. `https://deducteam.gitlabpages.inria.fr/nubo/index.html`, 2020. Accessed: 2022-29-11.

13 Alexis Dorra. Équivalence de Curry-Howard entre le lambda-Pi-calcul et la logique intuitionniste. Bachelor internship report, LIX École Polytechnique, 2011.

14 Gilles Dowek. On the definition of the classical connectives and quantifiers. *CoRR*, 2015. `arXiv:1601.01782`.

15 Gilles Dowek. Models and termination of proof reduction in the lambda pi-calculus modulo theory. In Ioannis Chatzigiannakis, Piotr Indyk, Fabian Kuhn, and Anca Muscholl, editors, *44th International Colloquium on Automata, Languages, and Programming, ICALP 2017, July 10-14, 2017, Warsaw, Poland*, volume 80 of *LIPIcs*, pages 109:1–109:14. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2017. `doi:10.4230/LIPIcs.ICALP.2017.109`.

16 Gilles Dowek, Thérèse Hardin, and Claude Kirchner. HOL-$\lambda\sigma$ an intentional first-order expression of higher-order logic. *Mathematical Structures in Computer Science*, 11(1):1–25, 2001.

17 Gilles Dowek and Benjamin Werner. Proof normalization modulo. *J. Symb. Log.*, 68(4):1289–1316, 2003. `doi:10.2178/jsl/1067620188`.

18 Thiago Felicissimo. Adequate and computational encodings in the logical framework dedukti. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction, FSCD 2022, August 2-5, 2022, Haifa, Israel*, volume 228 of *LIPIcs*, pages 25:1–25:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2022. `doi:10.4230/LIPIcs.FSCD.2022.25`.

19 Guillaume Genestier. Encoding Agda Programs Using Rewriting. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction (FSCD 2020)*, volume 167 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 31:1–31:17, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.FSCD.2020.31`.

20 Herman Geuvers. A short and flexible proof of strong normalization for the calculus of constructions. In Peter Dybjer, Bengt Nordström, and Jan M. Smith, editors, *Types for Proofs and Programs, International Workshop TYPES'94, Båstad, Sweden, June 6-10, 1994, Selected Papers*, volume 996 of *Lecture Notes in Computer Science*, pages 14–38. Springer, 1994. `doi:10.1007/3-540-60579-7_2`.

21 Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In Franz Baader and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 11th International Conference, LPAR 2004, Montevideo, Uruguay, March 14-18, 2005, Proceedings*, volume 3452 of *Lecture Notes in Computer Science*, pages 301–331. Springer, 2004. `doi:10.1007/978-3-540-32275-7_21`.

22 Jean-Yves Girard. On the unity of logic. *Ann. Pure Appl. Logic*, 59(3):201–217, 1993. `doi:10.1016/0168-0072(93)90093-S`.

23 Jean-Yves Girard, Paul Taylor, and Yves Lafont. *Proofs and types.* Cambridge Tracts in Theoretical Computer Science, 7, 1989.

24 Mohamed Yacine El Haddad, Guillaume Burel, and Frédéric Blanqui. EKSTRAKTO A tool to reconstruct dedukti proofs from TSTP files (extended abstract). In Giselle Reis and Haniel Barbosa, editors, *Proceedings Sixth Workshop on Proof eXchange for Theorem Proving, PxTP 2019, Natal, Brazil, August 26, 2019*, volume 301 of *EPTCS*, pages 27–35, 2019. `doi:10.4204/EPTCS.301.5`.

25 Robert Harper, Furio Honsell, and Gordon D. Plotkin. A framework for defining logics. In *Proceedings of the Symposium on Logic in Computer Science (LICS '87), Ithaca, New York, USA, June 22-25, 1987*, pages 194–204. IEEE Computer Society, 1987.

26 Gabriel Hondet and Frédéric Blanqui. Encoding of predicate subtyping with proof irrelevance in the λΠ-calculus modulo theory. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs, TYPES 2020, March 2-5, 2020, University of Turin, Italy*, volume 188 of *LIPIcs*, pages 6:1–6:18. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPIcs.TYPES.2020.6.

27 Alfred Horn. Logic with truth values in a linearly ordered heyting algebra. *J. Symb. Log.*, 34(3):395–408, 1969. doi:10.2307/2270905.

28 Jean-Pierre Jouannaud and Jianqi Li. Termination of Dependently Typed Rewrite Rules. In Thorsten Altenkirch, editor, *13th International Conference on Typed Lambda Calculi and Applications (TLCA 2015)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 257–272, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TLCA.2015.257.

29 Andrei Nikolaevich Kolmogorov. On the principle of the excluded middle. In *Matematicheskij Sbornik*, volume 32, pages 646–667, 1925.

30 C.L.M. Kop. *Higher Order Termination: Automatable Techniques for Proving Termination of Higher-Order Term Rewriting Systems*. PhD thesis, Vrije Universiteit Amsterdam, 2012. Naam instelling promotie: VU Vrije Universiteit Naam instelling onderzoek: VU Vrije Universiteit.

31 Keiichirou Kusakari and Masahiko Sakai. Enhancing dependency pair method using strong computability in simply-typed term rewriting. *Appl. Algebra Eng., Commun. Comput.*, 18(5):407–431, October 2007.

32 Chin Lee, Neil Jones, and Amir Ben-Amram. The size-change principle for program termination. *ACM SIGPLAN Notices*, 36, January 2001. doi:10.1145/360204.360210.

33 Chuck Liang and Dale Miller. Unifying classical and intuitionistic logics for computational control. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 283–292. IEEE Computer Society, 2013. doi:10.1109/LICS.2013.34.

34 Paqui Lucio. Structured sequent calculi for combining intuitionistic and classical first-order logic. In Hélène Kirchner and Christophe Ringeissen, editors, *Frontiers of Combining Systems, Third International Workshop, FroCoS 2000, Nancy, France, March 22-24, 2000, Proceedings*, volume 1794 of *Lecture Notes in Computer Science*, pages 88–104. Springer, 2000. doi:10.1007/10720084_7.

35 Paul-André Melliès and Benjamin Werner. A generic normalisation proof for pure type systems. In Eduardo Giménez and Christine Paulin-Mohring, editors, *Types for Proofs and Programs, International Workshop TYPES'96, Aussois, France, December 15-19, 1996, Selected Papers*, volume 1512 of *Lecture Notes in Computer Science*, pages 254–276. Springer, 1996. doi:10.1007/BFb0097796.

36 Luiz Carlos Pereira and Ricardo Oscar Rodriguez. Normalization, soundness and completeness for the propositional fragment of prawitz' ecumenical system. In *Revista Portuguesa De Filosofia 73, no. 3/4*, pages 1153–1168, 2017.

37 Elaine Pimentel, Luiz Carlos Pereira, and Valeria de Paiva. An ecumenical notion of entailment. *Synthese*, pages 1–23, 2019. doi:10.1007/s11229-019-02226-5.

38 Dag Prawitz. Classical versus intuitionnistic logic. In Edward Hermann Haeusler, Wagner de Campos Sanz, Bruno Lopes, and College Publications, editors, *Why is this a proof ? Festschrift for Luiz Carlos Pereira*, pages 15–32, 2015.

39 Raphaël Cauderlier. Sigmaid. https://gitlab.math.univ-paris-diderot.fr/cauderlier/sigmaid, 2014. Accessed: 2022-29-11.

40 Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique)*. PhD thesis, Mines ParisTech, France, 2015. URL: https://tel.archives-ouvertes.fr/tel-01299180.

41 William W. Tait. Intensional interpretations of functionals of finite type I. *J. Symb. Log.*, 32(2):198–212, 1967. doi:10.2307/2271658.

42 Terese. *Term rewriting systems*, volume 55 of *Cambridge tracts in theoretical computer science*. Cambridge University Press, 2003.

43 François Thiré. Sharing a Library between Proof Assistants: Reaching out to the HOL Family. *Electronic Proceedings in Theoretical Computer Science*, 274:57–71, July 2018. `doi:10.4204/EPTCS.274.5`.

44 François Thiré. *Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti)*. PhD thesis, École normale supérieure Paris-Saclay, Cachan, France, 2020. URL: `https://tel.archives-ouvertes.fr/tel-03224039`.

On the Fair Termination of Client-Server Sessions

Luca Padovani
University of Camerino, Italy

Abstract

Client-server sessions are based on a variation of the traditional interpretation of linear logic propositions as session types in which non-linear channels (those regulating the interaction between a pool of clients and a single server) are typed by *coexponentials* instead of the usual exponentials. Coexponentials enable the modeling of racing interactions, whereby clients compete to interact with a single server whose internal state (and thus the offered service) may change as the server processes requests sequentially. In this work we present a fair termination result for CSLL^∞, a core calculus of client-server sessions. We design a type system such that every well-typed term corresponds to a valid derivation in $\mu\mathsf{MALL}^\infty$, the infinitary proof theory of linear logic with least and greatest fixed points. We then establish a correspondence between reductions in the calculus and principal reductions in $\mu\mathsf{MALL}^\infty$. Fair termination in CSLL^∞ follows from cut elimination in $\mu\mathsf{MALL}^\infty$.

2012 ACM Subject Classification Theory of computation → Linear logic; Theory of computation → Process calculi; Theory of computation → Program analysis

Keywords and phrases client-server sessions, linear logic, fixed points, fair termination, cut elimination

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.5

Acknowledgements The author is grateful to the anonymous reviewers of the TYPES'22 post-proceedings for their helpful comments, observations and suggestions of related work.

1 Introduction

Session types [14, 15, 16] are descriptions of communication protocols enabling the static enforcement of a variety of safety and liveness properties, including the fact that communication channels are used according to their protocol (*fidelity*), that processes do not get stuck (*deadlock freedom*), that pending communications are eventually completed (*livelock freedom*), that sessions eventually end (*termination*). It is possible to trace a close correspondence between session types and propositions of linear logic, and between the typing rules of a session type system and the proof rules of linear logic [21, 6, 19]. This correspondence provides session type theories with a solid logical foundation and enables the application of known results concerning linear logic proofs into the domain of communicating protocols. One notable example is *cut elimination*: the fact that every linear logic proof can be reduced to a form that does not make use of the cut rule means that the process described by the proof can be reduced to a form in which no pending communication is present, provided that there is a good correspondence between cut reductions in proofs and reductions in processes.

The development of session type systems based on linear logic also poses some challenges with respect to their ability to cope with "real-world" scenarios. An example, which is the focus of this work, is the modeling of the interactions between a *pool of clients* and a *single server*. By definition, a server is a process that can handle an unbounded number of requests made by clients. In a session type system based on linear logic, it is natural to associate the channel from which a server accepts client requests with a type of the form $!T$, indicating the unlimited availability of a service with type T. In fact, it was observed early on [11] that the meaning of the "of course" modality $!T$ could be informally expressed by the equation

$$!T \cong \mathbf{1} \mathbin{\&} T \mathbin{\&} (!T \otimes !T)$$

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 5; pp. 5:1–5:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

which could be read "as many copies of T as the clients require". While appealing from a theoretical point of view, the association between the concept of server and the "of course" modality is both unrealistic and imprecise. First of all, it models the "unlimited" availability of the server by means of unlimited *parallel* copies of the server, each copy dealing with a single request, rather than by a *single* process that is capable of handling an unlimited number of requests *sequentially*. Second, it fails to capture the fact that each connection between a client and the server may alter the server's internal state, in such a way that different connections may potentially affect each other.

These considerations have led Qian et al. [20] to develop CSLL (for "Client-Server Linear Logic"), a session type system based on linear logic which includes the *coexponential* modalities $¿T$ and $¡T$ whose meaning can be (informally) expressed by the equations

$$¿T \cong \mathbf{1} \oplus T \oplus (¿T \otimes ¿T) \qquad \text{and} \qquad ¡T \cong \bot \mathbin{\&} T \mathbin{\&} (¡T \parr ¡T) \tag{1}$$

according to which a server that behaves as $¡T$ offers T as many times as necessary to satisfy all client requests, but it does so sequentially and in some (unspecified) order. Qian et al. [20] show that well-typed CSLL processes are deadlock free, but they leave a proof of termination to future work conjecturing that it could be quite involved. A proof of this property is valuable since termination (combined with deadlock freedom) implies livelock freedom.

In this paper we attack the problem of establishing a termination result for CSLL. Instead of providing an *ad hoc* proof, we attempt to reduce the termination problem for CSLL to the cut elimination property of a known logical system. To this aim, we propose a variation of CSLL called CSLL$^\infty$ that is in close relationship with μMALL$^\infty$ [3, 8, 2], the infinitary proof theory of multiplicative-additive linear logic with least and greatest fixed points. The basic idea is to encode the coexponentials in CSLL$^\infty$'s type system as fixed points in μMALL$^\infty$ following their expected meaning (Equation (1)). At this point, the cut elimination property of μMALL$^\infty$ should allow us to deduce that well-typed CSLL$^\infty$ processes do not admit infinite reduction sequences. As it turns out, we are unable to follow this plan of action in full. The problem is that some reductions in CSLL$^\infty$ do not correspond to cut reduction steps in μMALL$^\infty$. More specifically, even though clients are queued into client pools, they should be able to reduce in any order, independently of their position in the queue. This independent reduction of the clients in the same pool is not matched by the sequence of cut reduction steps that are performed in the cut elimination proof of μMALL$^\infty$. Still, the cut elimination property of μMALL$^\infty$ allows us to prove a useful result, namely that every well-typed CSLL$^\infty$ process is *fairly terminating.* Fair termination [13, 10] is weaker than termination since it does not rule out the existence of infinite reduction sequences. However, it guarantees that every *fair* and maximal reduction sequence of a well-typed CSLL$^\infty$ process is finite, under a suitable fairness assumption. In particular, fair termination is strong enough (when combined with deadlock freedom) to guarantee livelock freedom.

The adoption of μMALL$^\infty$ as logical foundation for CSLL$^\infty$ has another advantage. In the original presentation of CSLL [20] the process calculus is equipped with an unconventional operational semantics whereby reductions can occur underneath prefixes and prefixes may be moved around crossing restrictions, parallel compositions and other (unrelated) prefixes. This semantics is justified to keep the process reduction rules and the cut reduction rules sufficiently aligned, so that the cut elimination property in the logic can be reflected to some valuable property in the calculus, such as deadlock freedom. In contrast, CSLL$^\infty$ features an entirely conventional reduction semantics. We can afford to do so because μMALL$^\infty$ is an *infinitary* proof system in which the cut elimination property is proved bottom-up by *reducing outermost cuts first.* This reduction strategy matches the ordinary reduction

Table 1 Syntax of CSLL$^\infty$.

$$\begin{array}{rlll|lll}
P, Q ::= & \mathsf{A}\langle\overline{x}\rangle & \text{invocation} & & (x)(P \mid Q) & \text{parallel composition} & \\
\mid & \mathsf{fail}\, x & \text{failure} & & ¿x[] & \text{empty pool} & \\
\mid & \mathsf{wait}\, x.P & \text{wait} & & \mathsf{close}\, x & \text{close} & \\
\mid & x(y).P & \text{input} & & x[y](P \mid Q) & \text{output} & \\
\mid & \mathsf{case}\, x\{P, Q\} & \text{branch} & & \mathsf{in}_i\, x.P & \text{select} & i \in \{1,2\} \\
\mid & ¡x(y)\{P, Q\} & \text{server} & & ¿x[y].P :: Q & \text{client pool} &
\end{array}$$

semantics of any process calculus in which reductions happen at the outermost levels of processes. In the end, since the reduction semantics of CSLL$^\infty$ is stricter than that of CSLL, the deadlock freedom and the fair termination results we prove for CSLL$^\infty$ are somewhat stronger than their counterparts in the context of CSLL.

Structure of the paper. Section 2 describes syntax and semantics of CSLL$^\infty$ and defines the notion of fairly terminating process. We develop the type system for CSLL$^\infty$ in Section 3. In Section 4 we recall the key elements of μMALL$^\infty$, before addressing the proof that well-typed CSLL$^\infty$ processes fairly terminate in Section 5. Section 6 revisits an example of non-deterministic server given by Qian et al. [20] in our setting. We summarize our results and further compare CSLL$^\infty$ with CSLL [20] and other related work in Section 7. Some proofs and definitions have been moved into Appendix A.

2 Syntax and Semantics of CSLL$^\infty$

In this section we define syntax and semantics of CSLL$^\infty$, a calculus of sessions in which servers handle client requests sequentially. The syntax of CSLL$^\infty$ makes use of an infinite set $\mathcal{V}$ of *channels* ranged over by x, y and z and a set $\mathcal{P}$ of *process names* ranged over by A, B, and so on. In CSLL$^\infty$ channels are of two kinds (which will be distinguished by their type): *session channels* connect two communicating processes; *shared channels* connect an unbounded number of clients with a single server. The structure of terms is given by the grammar in Table 1 and their meaning is informally described below. The term $(x)(P \mid Q)$ represents the parallel composition of P and Q connected by the restricted channel x, which can be either a session channel or a shared channel. The term $\mathsf{fail}\, x$ represents a process that signals a failure on channel x. The term $\mathsf{close}\, x$ models the closing of a session, whereas $\mathsf{wait}\, x.P$ models a process that waits for x to be closed and then continues as P. The term $x[y](P \mid Q)$ models a process that creates a new channel y, sends y over x, uses y as specified by P and x as specified by Q. The term $x(y).P$ models a process that receives a channel y from x and then behaves as P. The term $\mathsf{in}_i\, x.P$ models a process that sends the label in_i over x and then behaves as P. In this work we only consider two labels in_1 and in_2, although it is common to allow for an arbitrary set of atomic labels. Dually, the term $\mathsf{case}\, x\{P_1, P_2\}$ models a process that waits for a label in_i from x and then behaves according to P_i. The term $¿x[]$ models the empty pool of clients connecting with a server on the shared channel x, whereas the term $¿x[y].P :: Q$ models a client pool consisting of a client that connects with a server on channel x and behaves as P and another client pool Q. Occasionally we write $¿x[y].P$ instead of $¿x[y].P :: ¿x[]$. The term $¡x(y)\{P, Q\}$ models a server that waits for connections on the shared channel x. If a new connection y is established, the server continues as P. If no clients are left connecting on x, the service on x is terminated and the process continues as Q. Finally, a term $\mathsf{A}\langle\overline{x}\rangle$ represents the invocation of the process named

Table 2 Structrual pre-congruence and reduction semantics of CSLL^∞.

[s-par-comm]	$(x)(P \mid Q) \preccurlyeq (x)(Q \mid P)$	
[s-pool-comm]	$\text{¿}x[y].P :: \text{¿}u[v].Q :: R \preccurlyeq \text{¿}u[v].Q :: \text{¿}x[y].P :: R$	
[s-par-assoc]	$(x)(P \mid (y)(Q \mid R)) \preccurlyeq (y)((x)(P \mid Q) \mid R)$	$x \in \mathsf{fn}(Q) \setminus \mathsf{fn}(R), y \notin \mathsf{fn}(P)$
[s-pool-par]	$\text{¿}x[y].P :: (z)(Q \mid R) \preccurlyeq (z)(\text{¿}x[y].P :: Q \mid R)$	$x \in \mathsf{fn}(Q), z \notin \mathsf{fn}(\text{¿}x[y].P)$
[s-par-pool]	$(z)(\text{¿}x[y].P :: Q \mid R) \preccurlyeq \text{¿}x[y].P :: (z)(Q \mid R)$	$z \notin \mathsf{fn}(\text{¿}x[y].P)$
[s-call]	$\mathsf{A}\langle\overline{x}\rangle \preccurlyeq P$	$\mathsf{A}(\overline{x}) \triangleq P$
[r-close]	$(x)(\mathsf{close}\, x \mid \mathsf{wait}\, x.P) \rightarrow P$	
[r-comm]	$(x)(x[y](P \mid Q) \mid x(y).R) \rightarrow (y)(P \mid (x)(Q \mid R))$	
[r-case]	$(x)(\mathsf{in}_i\, x.P \mid \mathsf{case}\, x\{Q_1, Q_2\}) \rightarrow (x)(P \mid Q_i)$	
[r-done]	$(x)(\text{¿}x[] \mid \text{¡}x(y)\{P, Q\}) \rightarrow Q$	
[r-connect]	$(x)(\text{¿}x[y].P :: Q \mid \text{¡}x(y)\{R_1, R_2\}) \rightarrow (y)(P \mid (x)(Q \mid R_1))$	
[r-par]	$(x)(P \mid R) \rightarrow (x)(Q \mid R)$	$P \rightarrow Q$
[r-pool]	$\text{¿}x[y].R :: P \rightarrow \text{¿}x[y].R :: Q$	$P \rightarrow Q$
[r-struct]	$P \rightarrow Q$	$P \preccurlyeq P' \rightarrow Q' \preccurlyeq Q$

A with arguments $\overline{x}$. We assume that each process name is associated with a unique global definition of the form $\mathsf{A}(\overline{x}) \triangleq P$. The notation $\overline{e}$ is used throughout the paper to represent possibly empty sequences $e_1, \dots, e_n$ of various entities.

The notions of free and bound names are defined in the expected way. Note that the output operations $x[y](P \mid Q)$ and $\text{¿}x[y].P :: Q$ bind y in P but not in Q. We write $\mathsf{fn}(P)$ and $\mathsf{bn}(P)$ for the sets of free and bound names in P, we identify processes up to renaming of bound channel names and we require $\mathsf{fn}(P) = \{\overline{x}\}$ for each global definition $\mathsf{A}(\overline{x}) \triangleq P$.

The operational semantics of CSLL^∞ is given by a structural precongruence relation $\preccurlyeq$ and a reduction relation $\rightarrow$, both defined in Table 2 and described below. Rules [s-par-comm] and [s-pool-comm] state the expected commutativity of parallel and pool compositions. In particular, [s-pool-comm] allows clients in the same queue to swap positions, modeling the fact that the order in which they connect to the server is not deterministic. Rule [s-par-assoc] models the associativity of parallel composition. The side conditions make sure that no channel is captured ($y \notin \mathsf{fn}(P)$) or left dangling ($x \notin \mathsf{fn}(R)$) and that parallel processes remain connected ($x \in \mathsf{fn}(Q)$). The rules [s-pool-par] and [s-par-pool] deal with the mixed associativity between parallel and pool compositions. The side conditions ensure that no bound name leaves its scope and that parallel processes remain connected. Finally, [s-call] unfolds a process invocation to its definition.

Concerning the reduction relation, rule [r-close] models the closing of a session, rule [r-comm] models the exchange of a channel and [r-case] that of a label. Rule [r-connect] models the connection of a client with a server, whereas [r-done] deals with the case in which there are no clients left. Finally, [r-par] and [r-pool] close reductions under parallel compositions and client pools whereas [r-struct] allows reductions up to structural pre-congruence.

Hereafter we write $\Rightarrow$ for the reflexive, transitive closure of $\rightarrow$, we write $P \rightarrow$ if $P \rightarrow Q$ for some Q and $P \not\rightarrow$ if not $P \rightarrow$. Later on we will also use a restriction of CSLL^∞ dubbed $\mathsf{CSLL}^\infty_{\mathsf{det}}$ whose reduction relation, denoted by $\rightarrow_{\mathsf{det}}$, is obtained by removing the rules [s-pool-comm], [s-pool-par], [s-par-pool] and [r-pool] (all those with "pool" in their name) from $\rightarrow$. In essence, $\mathsf{CSLL}^\infty_{\mathsf{det}}$ is a more deterministic version of CSLL^∞ in which clients are forced to connect and reduce in the order in which they appear in client pools. Also, clients are no longer allowed to cross restricted channels.

▶ **Example 1.** We illustrate the features of CSLL^∞ by modeling a pool of clients that compete to access a shared resource, represented as a simple *lock*. When one client manages to acquire the lock, meaning that it has gained access to the resource, it prevents other clients from accessing the resource until the resource is released. We model the lock with this definition:

$$\mathsf{Lock}(x, z) \triangleq \text{¡}x(y)\{\mathsf{wait}\, y.\mathsf{Lock}\langle x, z\rangle, \mathsf{close}\, z\}$$

The lock is a server waiting for connections on the shared channel x, whereas each user is a client of the lock connecting on x. When a connection is established, the server waits until the resource is released, which is signalled by the termination of the session y, and then makes itself available again to handle further requests.

The following process models the concurrent access to the lock by two clients:

$$(x)(\text{¿}x[u].\mathsf{close}\, u :: \text{¿}x[v].\mathsf{close}\, v :: \text{¿}x[] \mid \mathsf{Lock}\langle x, z\rangle)$$

The order in which requests are handled by Lock is non-deterministic because of [s-pool-comm]. In this oversimplified example the users are indistinguishable and so non-determinism does not prevent the system to be confluent. In Section 6 we will see a more interesting example in which confluence is lost. This kind of interaction is typeable in CSLL^∞ thanks to coexponentials, which enable the concurrent access to a shared resource. ⌟

We conclude this section by defining various termination properties of interest. A *run* of P is a (finite or infinite) sequence $(P_0, P_1, \dots)$ of processes such that $P = P_0$ and $P_i \to P_{i+1}$ whenever P_{i+1} is a term in the sequence. A run is *maximal* if it is infinite or if it is finite and its last term (say Q) cannot reduce any further (that is, $Q \not\to$). We say that P is *terminating* if every maximal run of P is finite. We say that P is *weakly terminating* if P has a maximal finite run. A run of P is *fair* if it contains finitely many weakly terminating processes. We say that P is *fairly terminating* if every fair run of P is finite. Note that a fairly terminating process may admit infinite runs, but these go through infinitely many weakly terminating states. In other words, these runs represent executions of the process in which termination is always within reach but also always avoided, as if the system or the process itself is conspiring against termination. For this reason, these runs are considered "uninteresting" as far as termination is concerned and the process is considered to be practically terminating.

A fundamental property of any fairness notion is the fact that every finite run of a process should be extendable to a maximal fair one. This property, called *feasibility* [1] or *machine closure* [18], holds for our fairness notion and follows immediately from the next proposition.

▶ **Proposition 2.** *Every process has at least one maximal fair run.*

Proof. For an arbitrary process P there are two possibilities. If P is weakly terminating, then there exists Q such that $P \Rightarrow Q \not\to$. From this sequence of reductions we obtain a maximal run of P that is fair since it is finite. If P is not weakly terminating, then $P \to$ and $P \Rightarrow Q$ implies that Q is not weakly terminating. In this case we can build an infinite run of P which is fair since it does not go through any weakly terminating process. ◀

The given notion of fair termination admits an alternative characterization that does not refer to fair runs. This characterization provides us with the key proof principle to show that well-typed CSLL^∞ processes fairly terminate (Section 5).

▶ **Theorem 3.** *P is fairly terminating iff $P \Rightarrow Q$ implies that Q is weakly terminating.*

Table 3 Typing rules for CSLL^∞.

$$\text{[CALL]}\quad \dfrac{P \vdash \overline{x : T}}{\mathsf{A}\langle\overline{x}\rangle \vdash \overline{x : T}}\ \mathsf{A}(\overline{x}) \triangleq P
\qquad
\text{[CUT]}\quad \dfrac{P \vdash \Gamma, x : T \qquad Q \vdash \Delta, x : T^{\perp}}{(x)(P \mid Q) \vdash \Gamma, \Delta}
\qquad
[\top]\quad \dfrac{}{\mathsf{fail}\,x \vdash \Gamma, x : \top}$$

$$\textbf{no rule for } \mathbf{0}
\qquad
[\bot]\quad \dfrac{P \vdash \Gamma}{\mathsf{wait}\,x.P \vdash \Gamma, x : \bot}
\qquad
[\mathbf{1}]\quad \dfrac{}{\mathsf{close}\,x \vdash x : \mathbf{1}}
\qquad
[\parr]\quad \dfrac{P \vdash \Gamma, y : T, x : S}{x(y).P \vdash \Gamma, x : T \parr S}$$

$$[\otimes]\quad \dfrac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : S}{x[y](P \mid Q) \vdash \Gamma, \Delta, x : T \otimes S}
\qquad
[\&]\quad \dfrac{P \vdash \Gamma, x : T \qquad Q \vdash \Gamma, x : S}{\mathsf{case}\,x\{P, Q\} \vdash \Gamma, x : T \mathbin{\&} S}
\qquad
[\oplus]\quad \dfrac{P \vdash \Gamma, x : T_i}{\mathsf{in}_i\,x.P \vdash \Gamma, x : T_1 \oplus T_2}$$

$$\text{[SERVER]}\quad \dfrac{P \vdash \Gamma, x : \text{¡}T, y : T \qquad Q \vdash \Gamma}{\text{¡}x(y)\{P, Q\} \vdash \Gamma, x : \text{¡}T}
\qquad
\text{[CLIENT]}\quad \dfrac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : \text{¿}T}{\text{¿}x[y].P :: Q \vdash \Gamma, \Delta, x : \text{¿}T}
\qquad
\text{[DONE]}\quad \dfrac{}{\text{¿}x[] \vdash x : \text{¿}T}$$

Proof. ($\Leftarrow$) Suppose by contradiction that $(P_0, P_1, \dots)$ is an infinite fair run of P and note that $P \Rightarrow P_i$ for every i. From the hypothesis we deduce that every P_i is weakly terminating. Then the run contains infinitely many weakly terminating processes, which is absurd by definition of fair run. ($\Rightarrow$) Suppose that $P \Rightarrow Q$. Then there is a finite run of P that ends in Q. By Proposition 2 there is a maximal fair run of Q. By concatenating these two runs we obtain a maximal fair run of P that contains Q. From the hypothesis we deduce that this run is finite. Since Q occurs in this run, we conclude that Q is weakly terminating. ◀

3 Type System

In this section we develop the type system of CSLL^∞. Types are defined thus:

$$\textbf{Type} \qquad T, S ::= \bot \mid \mathbf{1} \mid \top \mid \mathbf{0} \mid T \parr S \mid T \otimes S \mid T \mathbin{\&} S \mid T \oplus S \mid \text{¡}T \mid \text{¿}T$$

Types extend the usual constants and connectives of multiplicative-additive linear logic with the coexponentials $\text{¡}T$ and $\text{¿}T$ and, in the context of CSLL^∞, they describe how channels are used by processes. Positive types indicate output operations whereas negative types indicate input operations. In particular: $\mathbf{1}/\bot$ describe a session channel used for sending/receiving a session termination signal; $\mathbf{0}/\top$ describe a session channel used for sending/receiving an impossible (empty) message; $T \otimes S/T \parr S$ describe a session channel used for sending/receiving a channel of type T and then according to S; $T_1 \oplus T_2/T_1 \mathbin{\&} T_2$ describe a session channel used for sending/receiving a label in_i and then according to T_i; finally, $\text{¿}T/\text{¡}T$ describe a shared channel used for sending/receiving a connection message establishing a session of type T. Each type T has a *dual* $T^\perp$ obtained in the expected way. For example, we have $(\mathbf{1} \oplus T)^\perp = \bot \mathbin{\&} T^\perp$ and $(\text{¡}T)^\perp = \text{¿}T^\perp$.

The typing rules for CSLL^∞ are shown in Table 3. Typing judgments have the form $P \vdash \Gamma$ and relate a process P with a *context* Γ. Contexts are finite maps from channel names to types written as $\overline{x : T}$. We let Γ and Δ range over contexts, we write $\emptyset$ for the empty context, we write $\mathsf{dom}(\Gamma)$ for the domain of Γ, namely for the set of channel names for which there is an association in Γ, and we write Γ, Δ for the union of Γ and Δ when $\mathsf{dom}(\Gamma) \cap \mathsf{dom}(\Delta) = \emptyset$.

For the most part, the typing rules coincide with those of a standard session type system based on linear logic [21, 19]. In particular, $[\textsc{cut}]$, $[\top]$, $[\bot]$, $[\mathbf{1}]$, $[\parr]$, $[\otimes]$, $[\&]$ and $[\oplus]$ relate the standard proof rules of multiplicative-additive classical linear logic with the corresponding forms of CSLL^∞. The rule $[\textsc{call}]$ deals with process invocations $\mathsf{A}\langle\overline{x}\rangle$ by unfolding the global definition of A, noted as side condition to the rule. Rule $[\textsc{server}]$ deals with servers $\text{¡}x(y)\{P, Q\}$. The continuation P, which is the actual handler of incoming connections, must be well typed in a context enriched with the channel y resulting from the connection. Note that x is still present in the context and with the same type, meaning that P must *also* be able to handle any further connection on the shared channel x. The continuation Q, which models the behavior of the server once no more clients are connecting on x, is not supposed to use x any longer. Rule $[\textsc{client}]$ deals with non-empty client pools $\text{¿}x[y].P :: Q$. The client P is connecting with a server through a shared channel x and establishes a session y. The rest of the pool Q is using x in the same way. Rule $[\textsc{done}]$ deals with the empty pool of clients connecting on x.[1]

The typing rules are interpreted coinductively. Therefore, a judgment $P \vdash \Gamma$ is derivable if there is a possibly infinite typing derivation for it. The need for infinite typing derivations stems from the fact that we type process invocations by "unfolding" them to the process they represent, so this unfolding may go on forever in the case of recursive processes.

▶ **Example 4.** Let us consider once again the process definitions in Example 1. We derive

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{call}]}{\mathsf{wait}\,y.\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, y : \bot, z : \mathbf{1}}\,[\bot] \qquad \dfrac{}{\mathsf{close}\,z \vdash z : \mathbf{1}}\,[\mathbf{1}]}{\text{¡}x(y)\{\mathsf{wait}\,y.\mathsf{Lock}\langle x,z\rangle, \mathsf{close}\,z\} \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{server}]}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}\,[\textsc{call}]$$

showing that Lock is well typed. Note that the typing derivation is infinite since Lock is a recursive process. We can now obtain the following typing derivation

$$\dfrac{\dfrac{\dfrac{}{\mathsf{close}\,u \vdash u : \mathbf{1}}\,[\mathbf{1}] \qquad \dfrac{\dfrac{}{\mathsf{close}\,v \vdash v : \mathbf{1}}\,[\mathbf{1}] \qquad \dfrac{}{\text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{done}]}{\text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{client}]}{\text{¿}x[u].\mathsf{close}\,u :: \text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \vdash x : \text{¿}\mathbf{1}}\,[\textsc{client}] \qquad \dfrac{\vdots}{\mathsf{Lock}\langle x,z\rangle \vdash x : \text{¡}\bot, z : \mathbf{1}}}{(x)(\text{¿}x[u].\mathsf{close}\,u :: \text{¿}x[v].\mathsf{close}\,v :: \text{¿}x[\,] \mid \mathsf{Lock}\langle x,z\rangle) \vdash z : \mathbf{1}}\,[\textsc{cut}]$$

showing that the system as a whole is well typed. ⌟

Adopting an infinitary type system will make it easy to relate CSLL^∞ with $\mu\mathsf{MALL}^\infty$ (Section 5). However, we must be careful in that some infinite typing derivations allow us to type processes that are not weakly terminating, as illustrated in the next example.

▶ **Example 5** (non-terminating process)**.** Consider the process $\Omega \triangleq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega)$ which creates a session x, immediately closes it and then repeats the same behavior. Clearly, this process is not weakly terminating because it can only reduce thus:

$$\Omega \preccurlyeq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \to \Omega \preccurlyeq (x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \to \cdots$$

[1] There are some analogies between the typing rules for client pools and *coweakening*, *codereliction* and *cocontraction* in Differential Linear Logic (DiLL) [9], although the exact relationship between coexponentials and DiLL remains to be established. Quian et al. [20] provide a few more details.

Nonetheless, we are able to find the following (infinite) typing derivation for Ω.

$$\dfrac{\dfrac{\dfrac{}{\mathsf{close}\,x \vdash x : \mathbf{1}}\,[\mathbf{1}] \qquad \dfrac{\dfrac{\vdots}{\Omega \vdash \emptyset}\,[\textsc{call}]}{\mathsf{wait}\,x.\Omega \vdash x : \bot}\,[\bot]}{(x)(\mathsf{close}\,x \mid \mathsf{wait}\,x.\Omega) \vdash \emptyset}\,[\textsc{cut}]}{\Omega \vdash \emptyset}\,[\textsc{call}]$$

Since we aim at ensuring fair termination for well-typed processes, we must consider this derivation as *invalid*. ⌟

In order to rule out processes like Ω in Example 5, we identify a class of valid typing derivations as follows.

▶ **Definition 6** (valid typing derivation)**.** *A typing derivation is* valid *if every infinite branch in it goes through infinitely many applications of the rule* [SERVER] *concerning the same channel.*

This validity condition requires that every infinite branch of a typing derivation describes the behavior of a server willing to accept an unbounded number of connection requests. If we look back at the infinite typing derivation for the Lock process in Example 4, we see that it is valid according to Definition 6 since the only infinite branch in it goes through infinitely many applications of the rule [SERVER] concerning the very same shared channel x. On the contrary, the typing derivation in Example 5 is invalid since the infinite branch in it does not go through any application of [SERVER].

The fact that every infinite branch must go through infinitely many applications of [SERVER] *concerning the very same shared channel* is a subtle point. Without the specification that it is the *same* shared channel to be found infinitely often, it would be possible to obtain invalid typing derivations as illustrated by the next example.

▶ **Example 7.** Consider the definition

$$\Omega\text{-}\mathsf{Server}(x) \triangleq \text{¡}x(y)\{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle, (z)(\text{¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle)\}$$

describing a server that waits for connections on the shared channel x. After each request, the server makes itself available again for handling more requests by the recursive invocation $\Omega\text{-}\mathsf{Server}\langle x\rangle$. Once all requests have been processed, the server creates a new shared channel on which an analogous server operates. Using the typing rules in Table 3 we are able to find the following typing derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : \text{¡}\bot}\,[\textsc{call}]}{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : \text{¡}\bot, y : \bot}\,[\bot] \qquad \dfrac{\dfrac{}{\text{¿}z[] \vdash z : \text{¿}\mathbf{1}}\,[\textsc{done}] \quad \dfrac{\vdots}{\Omega\text{-}\mathsf{Server}\langle z\rangle \vdash z : \text{¡}\bot}\,[\textsc{call}]}{(z)(\text{¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle) \vdash \emptyset}\,[\textsc{cut}]}{\text{¡}x(y)\{\mathsf{wait}\,y.\Omega\text{-}\mathsf{Server}\langle x\rangle, (z)(\text{¿}z[] \mid \Omega\text{-}\mathsf{Server}\langle z\rangle)\} \vdash x : \text{¡}\bot}\,[\textsc{server}]}{\Omega\text{-}\mathsf{Server}\langle x\rangle \vdash x : \text{¡}\bot}\,[\textsc{call}]$$

Notice that the derivation bifurcates in correspondence of the application of [SERVER] and also that each sub-tree is infinite, since it contains an unfolding of the Ω-Server process. For this reason, the derivation contains (infinitely) many infinite branches, which are obtained by either "going left" or "going right" each time [SERVER] is encountered. Each of these infinite branches goes through an application of [SERVER] infinitely many times, as requested by

Definition 6. Also, any such branch that "goes right" finitely many times eventually ends up going through infinitely many applications of [SERVER] that concern the same channel. In contrast, any branch that "goes right" infinitely many times keeps going through applications of [SERVER] concerning new shared channels created in correspondence of the application of [CUT]. In conclusion, this typing derivation is invalid and rightly so, or else the diverging process $(x)(¿x[] \mid \Omega\text{-}\mathsf{Server}\langle x\rangle)$ would be well typed in the empty context. ⌟

We conclude this section by stating two key properties of the type system, starting from the fact that typing is preserved by structural pre-congruence and reductions.

▶ **Theorem 8.** *Let $P \mathrel{\mathcal{R}} Q$ where $\mathcal{R} \in \{\preccurlyeq, \rightarrow\}$. Then $P \vdash \Gamma$ implies $Q \vdash \Gamma$.*

Also, processes that are well typed in a context of the form $x : \mathbf{1}$ are deadlock free.

▶ **Theorem 9** (deadlock freedom)**.** *If $P \vdash x : \mathbf{1}$ then either $P \preccurlyeq \mathsf{close}\, x$ or $P \rightarrow_{\mathsf{det}}$.*

Note that Theorem 9 uses $\rightarrow_{\mathsf{det}}$ instead of $\rightarrow$ in order to state that P is able to reduce if it is not (structurally pre-congruent to) $\mathsf{close}\, x$. Recalling that $\rightarrow_{\mathsf{det}} \subseteq \rightarrow$, the deadlock freedom property ensured by Theorem 9 is slightly stronger than one would normally expect. This formulation will be necessary in Section 5 when proving the soundness of the type system. The proofs of Theorems 8 and 9 can be found in Appendix A.

▶ **Example 10** (forwarder)**.** Most session calculi based on linear logic include a form $x \leftrightarrow y$ whose typing rule $x \leftrightarrow y \vdash x : T, y : T^{\perp}$ corresponds to the axiom $\vdash T, T^{\perp}$ of linear logic. The form $x \leftrightarrow y$ is usually interpreted as a forwarder between the channels x and y and it is useful for example to model the output of a free channel $x\langle y\rangle.P$ as the term $x[z](y \leftrightarrow z \mid P)$. In this example we show that there is no need to equip CSLL^{∞} with a native form $x \leftrightarrow y$ since its behavior can be encoded as a well-typed CSLL^{∞} process. To this aim, we define a family Link_T of process definitions by induction on T as follows

$$
\begin{aligned}
\mathsf{Link}_{\perp}(x, y) &\triangleq \mathsf{wait}\, x.\mathsf{close}\, y \\
\mathsf{Link}_{\top}(x, y) &\triangleq \mathsf{fail}\, x \\
\mathsf{Link}_{T \parr S}(x, y) &\triangleq x(u).y[v](\mathsf{Link}_T\langle u, v\rangle \mid \mathsf{Link}_S\langle x, y\rangle) \\
\mathsf{Link}_{T \& S}(x, y) &\triangleq \mathsf{case}\, x\{\mathsf{in}_1\, y.\mathsf{Link}_T\langle x, y\rangle, \mathsf{in}_2\, y.\mathsf{Link}_S\langle x, y\rangle\} \\
\mathsf{Link}_{¡T}(x, y) &\triangleq ¡x(u)\{¿y[v].\mathsf{Link}_T\langle u, v\rangle :: \mathsf{Link}_{¡T}\langle x, y\rangle, ¿y[]\}
\end{aligned}
$$

with the addition of the definitions $\mathsf{Link}_T(x, y) \triangleq \mathsf{Link}_{T^{\perp}}\langle y, x\rangle$ for the positive type constructors. It is easy to build a typing derivation for the judgment $\mathsf{Link}_T\langle x, y\rangle \vdash x : T, y : T^{\perp}$. Also, every infinite branch in such derivation eventually loops through an invocation of the form $\mathsf{Link}_{¡S}\langle u, v\rangle$, which goes through an application of [SERVER] concerning the channel u. So, the derivation of $\mathsf{Link}_T\langle x, y\rangle \vdash x : T, y : T^{\perp}$ is valid and the process $\mathsf{Link}_T\langle x, y\rangle$ is well typed. ⌟

4 A quick recollection of $\mu\mathsf{MALL}^{\infty}$

In this section we recall the main elements of $\mu\mathsf{MALL}^{\infty}$ [8, 3, 2], the infinitary proof system of the multiplicative additive fragment of linear logic extended with least and greatest fixed points. The syntax of $\mu\mathsf{MALL}^{\infty}$ *pre-formulas* makes use of an infinite set of *propositional variables* ranged over by X and Y and is given by the grammar below:

$$
\textbf{Pre-formula} \qquad \varphi, \psi ::= X \mid \perp \mid \top \mid \mathbf{0} \mid \mathbf{1} \mid \varphi \parr \psi \mid \varphi \otimes \psi \mid \varphi \& \psi \mid \varphi \oplus \psi \mid \nu X.\varphi \mid \mu X.\varphi
$$

Table 4 Proof rules of μMALL$^\infty$ [3, 8, 2].

$$
\begin{array}{cccccc}
[\textsc{cut}] & [\top] & [\bot] & [\mathbf{1}] & [\parr] & [\otimes] \\
\dfrac{\vdash \Sigma, F \qquad \vdash \Theta, F^{\perp}}{\vdash \Sigma, \Theta} &
\dfrac{}{\vdash \Sigma, \top} &
\dfrac{\vdash \Sigma}{\vdash \Sigma, \bot} &
\dfrac{}{\vdash \mathbf{1}} &
\dfrac{\vdash \Sigma, F, G}{\vdash \Sigma, F \parr G} &
\dfrac{\vdash \Sigma, F \qquad \vdash \Theta, G}{\vdash \Sigma, \Theta, F \otimes G}
\end{array}
$$

$$
\begin{array}{cccc}
[\&] & [\oplus] & [\nu] & [\mu] \\
\dfrac{\vdash \Sigma, F \qquad \vdash \Sigma, G}{\vdash \Sigma, F \mathbin{\&} G} &
\dfrac{\vdash \Sigma, F_i}{\vdash \Sigma, F_1 \oplus F_2} &
\dfrac{\vdash \Sigma, F\{\nu X.F/X\}}{\vdash \Sigma, \nu X.F} &
\dfrac{\vdash \Sigma, F\{\mu X.F/X\}}{\vdash \Sigma, \mu X.F}
\end{array}
$$

The fixed point operators μ and ν are the binders of propositional variables and the notions of free and bound variables are defined accordingly. A μMALL$^\infty$ *formula* is a closed pre-formula. We write $\{\varphi/X\}$ for the capture-avoiding substitution of all free occurrences of X with φ and $\varphi^\perp$ for the *dual* of φ, which is the involution such that

$$X^\perp = X \qquad (\mu X.\varphi)^\perp = \nu X.\varphi^\perp \qquad (\nu X.\varphi)^\perp = \mu X.\varphi^\perp$$

among the other expected equations. Postulating that $X^\perp = X$ is not a problem since we will always dualize formulas, which do not contain free propositional variables.

We write $\preceq$ for the *subformula ordering*, that is the least partial order such that $\varphi \preceq \psi$ if φ is a subformula of ψ. For example, if $\varphi \overset{\mathsf{def}}{=} \mu X.\nu Y.(X \oplus Y)$ and $\psi \overset{\mathsf{def}}{=} \nu Y.(\varphi \oplus Y)$ we have $\varphi \preceq \psi$ and $\psi \not\preceq \varphi$. When Φ is a set of formulas, we write $\mathsf{min}\,\Phi$ for its $\preceq$-minimum formula if it is defined. Occasionally we let $\star$ stand for an arbitrary binary connective (one of $\oplus$, $\otimes$, $\&$, or $\parr$) and σ stand for an arbitrary fixed point operator (either μ or ν).

In μMALL$^\infty$ it is important to distinguish among different *occurrences* of the same formula in a proof derivation. To this aim, formulas are annotated with *addresses*. We assume an infinite set $\mathcal{A}$ of *atomic addresses*, $\mathcal{A}^\perp$ being the set of their duals such that $\mathcal{A} \cap \mathcal{A}^\perp = \emptyset$ and $\mathcal{A}^{\perp\perp} = \mathcal{A}$. We use a and b to range over elements of $\mathcal{A} \cup \mathcal{A}^\perp$. An *address* is a string aw where $w \in \{i, l, r\}^*$. The dual of an address is defined as $(aw)^\perp = a^\perp w$. We use α and β to range over addresses, we write $\sqsubseteq$ for the prefix relation on addresses and we say that α and β are *disjoint* if $\alpha \not\sqsubseteq \beta$ and $\beta \not\sqsubseteq \alpha$.

A *formula occurrence* (or simply occurrence) is a pair φ_α made of a formula φ and an address α. We use F and G to range over occurrences and we extend to occurrences several operations defined on formulas. In particular: we use logical connectives to compose occurrences so that $\varphi_{\alpha l} \star \psi_{\alpha r} \overset{\mathsf{def}}{=} (\varphi \star \psi)_\alpha$ and $\sigma X.\varphi_{\alpha i} \overset{\mathsf{def}}{=} (\sigma X.\varphi)_\alpha$; the dual of an occurrence is obtained by dualizing both its formula and its address, that is $(\varphi_\alpha)^\perp \overset{\mathsf{def}}{=} \varphi^\perp_{\alpha^\perp}$; occurrence substitution preserves the address in the type within which the substitution occurs, but forgets the address of the occurrence being substituted, that is $\varphi_\alpha\{\psi_\beta/X\} \overset{\mathsf{def}}{=} \varphi\{\psi/X\}_\alpha$.

We write $\overline{F}$ for the formula obtained by forgetting the address of F. Finally, we write $\rightsquigarrow$ for the least reflexive relation on types such that $F_1 \star F_2 \rightsquigarrow F_i$ and $\sigma X.F \rightsquigarrow F\{\sigma X.F/X\}$.

The proof rules of μMALL$^\infty$ are shown in Table 4, where Σ and Θ range over sets of occurrences written as $F_1, \dots, F_n$. The rules allow us to derive *sequents* of the form $\vdash \Sigma$ and are standard except for $[\nu]$, which *unfolds* a greatest fixed point just like $[\mu]$ does. Being an infinitary proof system, μMALL$^\infty$ rules are meant to be interpreted coinductively. That is, a sequent $\vdash \Sigma$ is derivable if there exists an *arbitrary* (finite or infinite) proof derivation whose conclusion is $\vdash \Sigma$. Without a *validity condition* on derivations, such proof system is notoriously unsound. μMALL$^\infty$'s validity condition requires every infinite branch of a derivation to be supported by the continuous unfolding of a greatest fixed point. In order to formalize this condition, we start by defining *threads*, which are sequences of occurrences.

▶ **Definition 11** (thread). *A* thread *of F is a (finite or infinite) sequence of occurrences $(F_0, F_1, \dots)$ such that $F_0 = F$ and $F_i \rightsquigarrow F_{i+1}$ whenever $i+1$ is a valid index of the sequence.*

Hereafter we use t to range over threads. For example, if we consider $\varphi \stackrel{\text{def}}{=} \mu X.(X \oplus \mathbf{1})$, we have that $t \stackrel{\text{def}}{=} (\varphi_a, (\varphi \oplus \mathbf{1})_{ai}, \varphi_{ail}, \dots)$ is an infinite thread of φ_a.

Among all threads, we are interested in finding those in which a ν-formula is unfolded infinitely often. These threads, called ν-threads, are precisely defined thus:

▶ **Definition 12** (ν-thread). *Let $t = (F_0, F_1, \dots)$ be an infinite thread, let $\overline{t}$ be the corresponding sequence $(\overline{F_0}, \overline{F_1}, \dots)$ of formulas and let $\mathsf{inf}(t)$ be the set of elements of $\overline{t}$ that occur infinitely often in $\overline{t}$. We say that t is a ν-*thread *if $\min \mathsf{inf}(t)$ is defined and is a ν-formula.*

If we consider the infinite thread t above, we have $\mathsf{inf}(t) = \{\varphi, \varphi \oplus \mathbf{1}\}$ and $\min \mathsf{inf}(t) = \varphi$, so t is *not* a ν-thread because φ is not a ν-formula. Consider instead $\varphi \stackrel{\text{def}}{=} \nu X.\mu Y.(X \oplus Y)$ and $\psi \stackrel{\text{def}}{=} \mu Y.(\varphi \oplus Y)$ and observe that ψ is the "unfolding" of φ. Now $t_1 \stackrel{\text{def}}{=} (\varphi_a, \psi_{ai}, (\varphi \oplus \psi)_{aii}, \varphi_{aiil}, \dots)$ is a thread of φ_a such that $\mathsf{inf}(t_1) = \{\varphi, \psi, \varphi \oplus \psi\}$ and we have $\min \mathsf{inf}(t_1) = \varphi$ because $\varphi \preceq \psi$, so t_1 is a ν-thread. If, on the other hand, we consider the thread $t_2 \stackrel{\text{def}}{=} (\varphi_a, \psi_{ai}, (\varphi \oplus \psi)_{aii}, \psi_{aiir}, (\varphi \oplus \psi)_{aiiri}, \dots)$ such that $\mathsf{inf}(t_2) = \{\psi, \varphi \oplus \psi\}$ we have $\min \mathsf{inf}(t_2) = \psi$ because $\psi \preceq \varphi \oplus \psi$, so t_2 is not a ν-thread. Intuitively, the $\preceq$-minimum formula among those that occur infinitely often in a thread is the outermost fixed point operator that is being unfolded infinitely often. It is possible to show that this minimum formula is always well defined [8]. If such minimum formula is a greatest fixed point operator, then the thread is a ν-thread. Note that a ν-thread is necessarily infinite.

Now we proceed by identifying threads along branches of proof derivations. To this aim, we provide a precise definition of *branch*.

▶ **Definition 13** (branch). *A* branch *of a proof derivation is a sequence $(\vdash \Sigma_0, \vdash \Sigma_1, \dots)$ of sequents such that $\vdash \Sigma_0$ occurs somewhere in the derivation and $\vdash \Sigma_{i+1}$ is a premise of the rule application that derives $\vdash \Sigma_i$ whenever $i+1$ is a valid index of the sequence.*

An infinite branch is valid if supported by a ν-thread that originates somewhere therein.

▶ **Definition 14.** *Let $\gamma = (\vdash \Sigma_0, \vdash \Sigma_1, \dots)$ be an infinite branch in a derivation. We say that γ is* valid *if there exists $I \subseteq \mathbb{N}$ such that $(F_i)_{i \in I}$ is a ν-thread and $F_i \in \Sigma_i$ for every $i \in I$.*

▶ **Definition 15.** *A μMALL$^\infty$ derivation is* valid *if so are its infinite branches.*

5 Fair Termination of CSLL$^\infty$

In this section we prove that well-typed CSLL$^\infty$ processes fairly terminate. We do so by appealing to the alternative characterization of fair termination given by Theorem 3. Using that characterization and using the fact that typing is preserved by reductions (Theorem 8), it suffices to show that well-typed CSLL$^\infty$ processes *weakly* terminate. To do that, we encode a well-typed CSLL$^\infty$ process P into a (valid) μMALL$^\infty$ proof and we use the cut elimination property of μMALL$^\infty$ to argue that P has a finite maximal run.

Encoding of types

The encoding of CSLL$^\infty$ types into μMALL$^\infty$ formulas is the map $[\![\cdot]\!]$ defined by

$$[\![¿T]\!] = \mu X.(\mathbf{1} \oplus ([\![T]\!] \otimes X)) \qquad [\![¡T]\!] = \nu X.(\bot \,\&\, ([\![T]\!] \parr X)) \tag{2}$$

and extended homomorphically to all the other type constructors, which are in one-to-one correspondence with the connectives and constants of $\mu\mathsf{MALL}^\infty$. Notice that the image of the encoding is a relatively small subset of $\mu\mathsf{MALL}^\infty$ formulas in which different fixed point operators are never intertwined. Also notice that the encoding of the coexponentials does not follow exactly their expansion in Equation (1). Basically, we choose to interpret $¿T$ as a *list* of clients rather than as a *tree* of clients, following to the intuition that clients are queued when connecting to a server. The interpretation of ¡ follows as a consequence, as we want it to be the dual of the interpretation of ¿. Note that this interpretation of the coexponential modalities is the same used by Qian et al. [20].

Encoding of typing contexts

The next step is the encoding of CSLL^∞ contexts into $\mu\mathsf{MALL}^\infty$ sequents. Recall that a $\mu\mathsf{MALL}^\infty$ sequent is a set of occurrences and that an occurrence is a pair φ_α made of a formula φ and an address α. In order to associate addresses with formulas, we parametrize the encoding of CSLL^∞ contexts with an injective map σ from CSLL^∞ channels to addresses, since channels in (the domain of a) CSLL^∞ context uniquely identify the occurrence of a type (and thus of a formula). We write $x \mapsto \alpha$ for the singleton map that associates x with the address α and σ_1, σ_2 for the union of σ_1 and σ_2 when they have disjoint domains and codomains. Now, the encoding of a CSLL^∞ context is set of formulas defined by

$$[\![x_1 : T_1, \dots, x_n : T_n]\!]_{\sigma, x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n} \overset{\mathsf{def}}{=} [\![T_1]\!]_{\alpha_1}, \dots, [\![T_n]\!]_{\alpha_n}$$

Encoding of typing derivations

Just like for the encoding of CSLL^∞ contexts, also the encoding of typing derivations is parametrized by a map σ from CSLL^∞ channels to addresses. In addition, we also have to take into account the possibility that *restricted channels* are introduced in a CSLL^∞ context, which happens in the rule [CUT] of Table 3. The formula occurrence corresponding to the type of this newly introduced channel must have an address that is disjoint from that of any other occurrence. To guarantee this disjointness, we parametrize the encoding of CSLL^∞ derivations by an *infinite stream* ρ of pairwise distinct atomic addresses. Formally, ρ is an injective function $\mathbb{N} \to \mathcal{A}$. We write $a\rho$, $\mathsf{even}(\rho)$ and $\mathsf{odd}(\rho)$ for the streams defined by

$$(a\rho)(0) \overset{\mathsf{def}}{=} a \qquad (a\rho)(n+1) \overset{\mathsf{def}}{=} \rho(n) \qquad \mathsf{even}(\rho)(n) \overset{\mathsf{def}}{=} \rho(2n) \qquad \mathsf{odd}(\rho)(n) \overset{\mathsf{def}}{=} \rho(2n+1)$$

respectively. In words, $a\rho$ is the stream of atomic addresses that starts with a and continues as ρ whereas $\mathsf{even}(\rho)$ and $\mathsf{odd}(\rho)$ are the sub-streams of ρ consisting of addresses with an even (respectively, odd) index.

The encoding of a CSLL^∞ typing derivation is coinductively defined by a map $[\![\cdot]\!]_\sigma^\rho$ which we describe using the following notation. For every typing rule in Table 3

$$\frac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_n}{\mathcal{J}}\,[\text{RULE}] \qquad \text{we write} \qquad \left[\!\!\left[\frac{\mathcal{J}_1 \qquad \cdots \qquad \mathcal{J}_n}{\mathcal{J}}\right]\!\!\right]_\sigma^\rho = \pi$$

meaning that π is the $\mu\mathsf{MALL}^\infty$ derivation resulting from the encoding of the CSLL^∞ derivation for the judgment $\mathcal{J}$ in which the last rule is an application of [RULE]. Within π there will be instances of the $[\![\mathcal{J}_i]\!]_{\sigma_i}^{\rho_i}$ for suitable σ_i and ρ_i standing for the encodings of the CSLL^∞ sub-derivations for the judgments $\mathcal{J}_i$ that we find as premises of [RULE].

There is a close correspondence between many CSLL^∞ typing rules and $\mu\mathsf{MALL}^\infty$ proof rules so we only detail a few interesting cases of the encoding, starting from the typing rules $[\otimes]$ and $[⅋]$. A $\mu\mathsf{MALL}^\infty$ typing derivation ending with an application of these rules is encoded as follows:

$$\left\llbracket \frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : S}{x[y](P \mid Q) \vdash \Gamma, \Delta, x : T \otimes S} \right\rrbracket^{\rho}_{\sigma, x \mapsto \alpha} = \frac{\llbracket P \vdash \Gamma, y : T \rrbracket^{\mathsf{even}(\rho)}_{\sigma, y \mapsto \alpha l} \qquad \llbracket Q \vdash \Delta, x : S \rrbracket^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto \alpha r}}{\vdash \llbracket \Gamma, \Delta \rrbracket_{\sigma}, \llbracket T \otimes S \rrbracket_{\alpha}}\,[\otimes]$$

$$\left\llbracket \frac{P \vdash \Gamma, y : T, x : S}{x(y).P \vdash \Gamma, x : T ⅋ S} \right\rrbracket^{\rho}_{\sigma, x \mapsto \alpha} = \frac{\llbracket P \vdash \Gamma, y : T, x : S \rrbracket^{\rho}_{\sigma, y \mapsto \alpha l, x \mapsto \alpha r}}{\vdash \llbracket \Gamma \rrbracket_{\sigma}, \llbracket T ⅋ S \rrbracket_{\alpha}}\,[⅋]$$

Notice that the types $T \otimes S$ and $T ⅋ S$ associated with x in the conclusion of the rules are encoded into the occurrences $\llbracket T \otimes S \rrbracket_\alpha$ and $\llbracket T ⅋ S \rrbracket_\alpha$ where α is the address associated with x in $\sigma, x \mapsto \alpha$. This address is suitably updated in the encoding of the premises of the rules. In the case of $[\otimes]$, the original stream ρ of atomic addresses is split into two disjoint streams in the encoding of the premises to ensure that no atomic address is used twice.

Every application of [call] is simply erased in the encoding:

$$\left\llbracket \frac{P \vdash \overline{x : T}}{\mathsf{A}\langle \overline{x} \rangle \vdash \overline{x : T}} \right\rrbracket^{\rho}_{\sigma} = \llbracket P \vdash \overline{x : T} \rrbracket^{\rho}_{\sigma}$$

The validity of the CSLL^∞ typing derivation guarantees that there cannot be an infinite chain of process invocations in a well-typed process. A proof of this fact is given by Lemma 21 in Appendix A. For this reason, the encoding of CSLL^∞ derivations is well defined despite the fact that applications of [call] are erased.

Another case worth discussing is that of the rule [cut], which is handled as follows:

$$\left\llbracket \frac{P \vdash \Gamma, x : T \qquad Q \vdash \Delta, x : T^{\perp}}{(x)(P \mid Q) \vdash \Gamma, \Delta} \right\rrbracket^{a\rho}_{\sigma} = \frac{\llbracket P \vdash \Gamma, x : T \rrbracket^{\mathsf{even}(\rho)}_{\sigma, x \mapsto a} \qquad \llbracket Q \vdash \Delta, x : T^{\perp} \rrbracket^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto a^{\perp}}}{\vdash \llbracket \Gamma, \Delta \rrbracket_{\sigma}}\,[\text{cut}]$$

The first address from the infinite stream $a\rho$, which is guaranteed to be distinct from any other address used so far and that will be used in the rest of the encoding, is associated with the newly introduced variable x. Similarly to the case of $[\otimes]$, the tail of the stream is split in the encoding of the two premises of [cut] so as to preserve this guarantee.

We now consider the applications of [done], [client] and [server] which account for the most relevant part of the encoding. These rule applications are encoded by considering the interpretation of the co-exponentials in terms of least and greatest fixed points (Equation (2)) and then by applying the suitable $\mu\mathsf{MALL}^\infty$ proof rules ($[\mu]$ and $[\nu]$ in particular). We have

$$\left\llbracket \frac{}{{¿}x[\,] \vdash x : {¿}T} \right\rrbracket^{\rho}_{\sigma, x \mapsto \alpha} = \frac{\dfrac{\dfrac{}{\vdash \mathbf{1}_{\alpha i l}}\,[\mathbf{1}]}{\vdash \llbracket \mathbf{1} \oplus (T \otimes {¿}T) \rrbracket_{\alpha i}}\,[\oplus]}{\vdash \llbracket {¿}T \rrbracket_{\alpha}}\,[\mu]$$

for the applications of [done] and

$$\left\llbracket \frac{P \vdash \Gamma, y : T \qquad Q \vdash \Delta, x : \text{¿}T}{\text{¿}x[y].P :: Q \vdash \Gamma, \Delta, x : \text{¿}T} \right\rrbracket^{\rho}_{\sigma, x \mapsto \alpha} =$$

$$\dfrac{\dfrac{\dfrac{\llbracket P \vdash \Gamma, y : T \rrbracket^{\mathsf{even}(\rho)}_{\sigma, y \mapsto \alpha irl} \qquad \llbracket Q \vdash \Delta, x : \text{¿}T \rrbracket^{\mathsf{odd}(\rho)}_{\sigma, x \mapsto \alpha irr}}{\vdash \llbracket \Gamma, \Delta \rrbracket_{\sigma}, \llbracket T \otimes \text{¿}T \rrbracket_{\alpha ir}}\;[\otimes]}{\vdash \llbracket \Gamma, \Delta \rrbracket_{\sigma}, \llbracket \mathbf{1} \oplus (T \otimes \text{¿}T) \rrbracket_{\alpha i}}\;[\oplus]}{\vdash \llbracket \Gamma, \Delta \rrbracket_{\sigma}, \llbracket \text{¿}T \rrbracket_{\alpha}}\;[\mu]$$

for the applications of [CLIENT]. Finally, the applications of [SERVER] are encoded thus:

$$\left\llbracket \frac{P \vdash \Gamma, x : \text{¡}T, y : T \qquad Q \vdash \Gamma}{\text{¡}x(y)\{P, Q\} \vdash \Gamma, x : \text{¡}T} \right\rrbracket^{\rho}_{\sigma, x \mapsto \alpha} =$$

$$\dfrac{\dfrac{\dfrac{\llbracket Q \vdash \Gamma \rrbracket^{\rho}_{\sigma}}{\vdash \llbracket \Gamma \rrbracket_{\sigma}, \bot_{\alpha il}}\;[\bot] \qquad \dfrac{\llbracket P \vdash \Gamma, x : \text{¡}T, y : T \rrbracket^{\rho}_{\sigma, x \mapsto \alpha irr, y \mapsto \alpha irl}}{\vdash \llbracket \Gamma \rrbracket_{\sigma}, \llbracket T \parr \text{¡}T \rrbracket_{\alpha ir}}\;[\parr]}{\vdash \llbracket \Gamma \rrbracket_{\sigma}, \llbracket \bot \mathbin{\&} (T \parr \text{¡}T) \rrbracket_{\alpha i}}\;[\&]}{\vdash \llbracket \Gamma \rrbracket_{\sigma}, \llbracket \text{¡}T \rrbracket_{\alpha}}\;[\nu]$$

Note that in this last case it is not necessary to split the stream ρ since the P and Q branches of the server are mutually exclusive (the reduction rules [R-CONNECT] and [R-DONE] pick one or the other branch, but not both). A similar thing happens in the encoding of the applications of [&], not shown here.

Validity of encoded typing derivations

Now that we have shown how every CSLL^{∞} typing derivation is encoded into a $\mu\mathsf{MALL}^{\infty}$ derivation, we argue that the encoding preserves validity. More specifically, a valid CSLL^{∞} typing derivation (Definition 6) is encoded into a valid $\mu\mathsf{MALL}^{\infty}$ derivation (Definition 15). To see that this is the case, first observe that there is a one-to-one correspondence between the infinite branches in the two derivations. From Definition 6 we know that every infinite branch in a CSLL^{∞} derivation contains infinitely many applications of [SERVER] concerning the same shared channel x having type $\text{¡}T$ for some T. In the encoded derivation, this translates to the existence of a formula $\llbracket \text{¡}T \rrbracket$ that occurs infinitely often in the sequents making up this infinite branch. Now, suppose that the first occurrence of this formula is associated with some address α. From the encoding of [SERVER] we can then build the thread

$$t \overset{\mathsf{def}}{=} (\llbracket \text{¡}T \rrbracket_{\alpha}, \llbracket \bot \mathbin{\&} (T \parr \text{¡}T) \rrbracket_{\alpha i}, \llbracket T \parr \text{¡}T \rrbracket_{\alpha ir}, \llbracket \text{¡}T \rrbracket_{\alpha irr}, \dots)$$

which is *infinite*. Also note that $\mathsf{inf}(t) = \{\llbracket \text{¡}T \rrbracket, \llbracket \bot \mathbin{\&} (T \parr \text{¡}T) \rrbracket, \llbracket T \parr \text{¡}T \rrbracket\}$, that $\min \mathsf{inf}(t) = \llbracket \text{¡}T \rrbracket$, and that $\llbracket \text{¡}T \rrbracket$ is a ν-formula by Equation (2). In conclusion, t is a ν-thread (Definition 12) as required by the validity condition for $\mu\mathsf{MALL}^{\infty}$ pre-proofs (Definition 15).

Soundness of the type system

Now that we know how to obtain a $\mu\mathsf{MALL}^{\infty}$ proof from a well-typed CSLL^{∞} process we observe that each reduction rule of $\mathsf{CSLL}^{\infty}_{\mathsf{det}}$ corresponds to *one or more* principal reductions in a $\mu\mathsf{MALL}^{\infty}$ proof [8, Figure 3.2]. In particular, the reductions [R-CLOSE], [R-COMM] and [R-CASE] correspond to *exactly one* principal reduction in $\mu\mathsf{MALL}^{\infty}$ (for $\mathbf{1}/\bot$, $\otimes/\parr$ and $\oplus/\&$

respectively), whereas [r-done] and [t-client] correspond to *three* subsequent principal reductions in μMALL$^{\infty}$. For example, [r-connect] corresponds to the principal reduction μ/ν followed by $\oplus/\&$ followed by $\otimes/\parr$. Using this correspondence between CSLL$^{\infty}_{\mathsf{det}}$ and μMALL$^{\infty}$, we can prove that every CSLL$^{\infty}$ process that is well typed in a context of the form $x : \mathbf{1}$ is weakly terminating. Note that this correspondence holds for $\rightarrow_{\mathsf{det}}$ but *not* for $\rightarrow$ in general. However, since $\rightarrow_{\mathsf{det}} \subseteq \rightarrow$, this is enough to establish the weak termination of well-typed CSLL$^{\infty}$ processes in the general case.

▶ **Theorem 16.** *If $P \vdash x : \mathbf{1}$ then P is weakly terminating.*

Proof. Let $a\rho$ be an infinite stream of pairwise distinct atomic addresses. Every deterministic reduction of P (that is, according to $\rightarrow_{\mathsf{det}}$) can be mimicked by one or more principal reductions in the μMALL$^{\infty}$ proof $[\![P \vdash x : \mathbf{1}]\!]^{\rho}_{x \mapsto a}$. We know that μMALL$^{\infty}$ enjoys cut elimination [8]. In particular, there cannot be an infinite sequence of principal reductions in a μMALL$^{\infty}$ proof [8, Proposition 3.5]. It follows that there is no infinite sequence of deterministic reductions starting from P (using the CSLL$^{\infty}_{\mathsf{det}}$ semantics), that is $P \Rightarrow_{\mathsf{det}} Q \nrightarrow_{\mathsf{det}}$ for some Q. From Theorem 8 we deduce $Q \vdash x : \mathbf{1}$ and from Theorem 9 we deduce $Q \preccurlyeq \mathsf{close}\,x$. We conclude $P \Rightarrow\preccurlyeq \mathsf{close}\,x \nrightarrow$. In other words, P is weakly terminating. ◀

▶ **Corollary 17.** *If $P \vdash x : \mathbf{1}$ then P is fairly terminating.*

Proof. Straightforward consequence of Theorems 3 and 16. ◀

6 Example: a Compare-and-Swap register

In this section we illustrate a more complex scenario of client-server interaction that hightlights not only the fact that the server handles connections sequentially in an unspecified order but also the fact that each connection may change the server's internal state and affect other connections. More specifically, we show a modeling of the Compare-and-Swap (CAS) register of Qian et al. [20] in CSLL$^{\infty}$. A CAS register holds a boolean value true or false and is represented as a server that accepts connections from clients. Each client sends two boolean values to the server, an *expected value* and a *desired value*. If the expected value matches the content of the register, then the register is overwritten with the desired value. Otherwise, the register remains unchanged. We model boolean values as choices made in some session y. For instance, we can model the sending of true on y by the selection $\mathsf{in}_1\,y$ and the sending of false on y by the selection $\mathsf{in}_2\,y$. In fact, in this section we write true and false as aliases for the labels in_1 and in_2, respectively.

Below are two definitions for clients that differ for the expected and desired values they send to the CAS register:

$$\mathsf{Client}_{\mathsf{true},\mathsf{false}}(y) \triangleq \mathsf{true}\,y.\mathsf{false}\,y.\mathsf{close}\,y \qquad \mathsf{Client}_{\mathsf{false},\mathsf{true}}(y) \triangleq \mathsf{false}\,y.\mathsf{true}\,y.\mathsf{close}\,y$$

It is easy to see that both definitions are well typed. In particular, we can derive $\mathsf{Client}_{b,c}\langle y\rangle \vdash y : (\mathbf{1} \oplus \mathbf{1}) \oplus (\mathbf{1} \oplus \mathbf{1})$ for every $b, c \in \{\mathsf{true}, \mathsf{false}\}$ with two applications of [$\oplus$] and one application of [$\mathbf{1}$]. We combine two clients in a single pool as by the following definition

$$\mathsf{Clients}(x) \triangleq ¿x[y].\mathsf{Client}_{\mathsf{true},\mathsf{false}}\langle y\rangle :: ¿x[y].\mathsf{Client}_{\mathsf{false},\mathsf{true}}\langle y\rangle :: ¿x[]$$

for which we derive $\mathsf{Clients}\langle x\rangle \vdash x : ¿((\mathbf{1} \oplus \mathbf{1}) \oplus (\mathbf{1} \oplus \mathbf{1}))$ using [client] and [done].

For the CAS server we provide two definitions $\mathsf{CAS}_{\mathsf{true}}$ and $\mathsf{CAS}_{\mathsf{false}}$ corresponding to the states in which the register holds the value true and false, respectively.

$$\begin{aligned}
\mathsf{CAS}_{\mathsf{true}}(x,z) \triangleq \text{¡}x(y)\{&\mathsf{case}\,y\{\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\},\\
&\qquad\qquad\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle\}\},\\
&\mathsf{true}\,z.\mathsf{close}\,z\}\\
\mathsf{CAS}_{\mathsf{false}}(x,z) \triangleq \text{¡}x(y)\{&\mathsf{case}\,y\{\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\},\\
&\qquad\qquad\mathsf{case}\,y\{\mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle, \mathsf{wait}\,y.\mathsf{CAS}_{\mathsf{false}}\langle x,z\rangle\}\},\\
&\mathsf{false}\,z.\mathsf{close}\,z\}
\end{aligned}$$

The server in state $b \in \{\mathsf{true}, \mathsf{false}\}$ waits for connections on the shared channel x. If there is no client, the server sends b on z and terminates. If a client connects, then a session y is established. At this stage the server performs two input operations to receive the expected and desired values from the client. If the expected value does *not* match b, then the desired value is ignored and the server recursively invokes itself in the same state b. If the expected value matches b, then the server recursively invokes itself in a state that matches the client's desired value.

It is not difficult to obtain derivations for the judgments $\mathsf{CAS}_b\langle x,z\rangle \vdash x : \text{¡}((\bot \mathbin{\&} \bot) \mathbin{\&} (\bot \mathbin{\&} \bot)), z : \mathbf{1} \oplus \mathbf{1}$ for every $b \in \{\mathsf{true}, \mathsf{false}\}$. These derivations are valid since every infinite branch in them goes through an application of [SERVER] concerning the channel x. In conclusion, the CAS server is well typed and so is the composition $(x)(\mathsf{Clients}\langle x\rangle \mid \mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle)$.

Note that the process $(x)(\mathsf{Clients}\langle x\rangle \mid \mathsf{CAS}_{\mathsf{true}}\langle x,z\rangle)$ is not deterministic since it may reduce to either $\mathsf{true}\,z.\mathsf{close}\,z$ or $\mathsf{false}\,z.\mathsf{close}\,z$ depending on the order in which clients connect. Indeed, if $\mathsf{Client}_{\mathsf{true},\mathsf{false}}$ connects first, then the state of the register changes from true to false and then the connection with the second client changes it back from false to true. If, on the other hand, $\mathsf{Client}_{\mathsf{false},\mathsf{true}}$ connects first (because [S-POOL-COMM] is used), then the initial state of the register does not change and then it is changed from true to false when the client $\mathsf{Client}_{\mathsf{true},\mathsf{false}}$ finally connects.

7 Concluding Remarks

CSLL [20] is a non-deterministic session calculus based on linear logic in which servers handle multiple client requests sequentially. In this work we have targeted the problem of proving the termination of well-typed CSLL processes. To this aim, we have introduced CSLL^∞, a variant of CSLL closely related to $\mu\mathsf{MALL}^\infty$ [3, 8, 2], the infinitary proof system for multiplicative additive linear logic with fixed points. We have shown that well-typed CSLL^∞ processes are fairly terminating by encoding CSLL^∞ typing derivations into $\mu\mathsf{MALL}^\infty$ proofs and using the cut elimination property of $\mu\mathsf{MALL}^\infty$. Although fair termination is weaker than termination, it is strong enough to imply livelock freedom, which was one of the motivations for proving termination in the original CSLL work [20]. In our work, fair termination is termination under the fairness assumption that termination is not avoided forever (Theorem 3). However, inspection of our proof (Section 5) reveals that the fairness assumption can be substantially weakened: the fair termination in CSLL^∞ is reduced to the termination in $\mathsf{CSLL}^\infty_{\mathsf{det}}$, meaning that fair termination in CSLL^∞ is guaranteed if client requests are handled in order.

CSLL^∞ differs from the original CSLL in a few ways. In the interest of simplicity, we have chosen to omit constructs for modeling (pools of) sequential clients and replicated servers which are meant to be typed using the traditional exponential modalities. These features are orthogonal to the ones we are interested in and we think that they can be accommodated without substantial challenges following the same technical development illustrated in the present paper. In fact, the general support to fixed points in $\mu\mathsf{MALL}^\infty$ allows for this and other extensions, such as (co)recursive session types [19, 7]. Another difference is that CSLL^∞ adopts a reduction semantics that is completely ordinary for a process calculus. In particular, reductions are *not* allowed under prefixes, restrictions can*not* be moved beyond prefixes and (unrelated) prefixes can*not* be swapped. Nonetheless, we are able to relate the reduction semantics of CSLL^∞ with the cut reduction strategy of $\mu\mathsf{MALL}^\infty$ since $\mu\mathsf{MALL}^\infty$ proofs, which can be infinite, are reduced bottom-up. For this reason, we find that $\mu\mathsf{MALL}^\infty$ provides a natural logical foundation for session calculi based on linear logic.

Just like CSLL, also CSLL^∞ is related to SILL_S [4, 5] and HCP_ND [17], two session calculi based on linear logic that allow for races and non-determinism. In SILL_S, sessions can be *shared* among more than two communicating processes. Access to a shared session is regulated by means of explicit acquire/release actions that manifest themselves as special modalities in session types. The flexibility gained by session sharing may compromise deadlock freedom, which can be recovered by means of additional type structure [5]. HCP_ND uses bounded exponentials [12] to implement client/server interactions in which the amount of channel sharing is known (and bounded) in advance. Neither CSLL nor CSLL^∞ require such bounds. For example, the forwarder process $\mathsf{Link}_{¡T}\langle x, y\rangle$ in Example 10 would be ill typed in HCP_ND since the number of clients that may be willing to connect on x is not known *a priori*.

References

1 Krzysztof R. Apt, Nissim Francez, and Shmuel Katz. Appraising fairness in languages for distributed programming. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages, Munich, Germany, January 21-23, 1987*, pages 189–198. ACM Press, 1987. doi:10.1145/41625.41642.

2 David Baelde, Amina Doumane, Denis Kuperberg, and Alexis Saurin. Bouncing threads for circular and non-wellfounded proofs. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS, 2022*, 2022. arXiv:2005.08257.

3 David Baelde, Amina Doumane, and Alexis Saurin. Infinitary proof theory: the multiplicative additive case. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic, CSL 2016, August 29 – September 1, 2016, Marseille, France*, volume 62 of *LIPIcs*, pages 42:1–42:17. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2016. doi:10.4230/LIPIcs.CSL.2016.42.

4 Stephanie Balzer and Frank Pfenning. Manifest sharing with session types. *Proc. ACM Program. Lang.*, 1(ICFP):37:1–37:29, 2017. doi:10.1145/3110281.

5 Stephanie Balzer, Bernardo Toninho, and Frank Pfenning. Manifest deadlock-freedom for shared session types. In Luís Caires, editor, *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings*, volume 11423 of *Lecture Notes in Computer Science*, pages 611–639. Springer, 2019. doi:10.1007/978-3-030-17184-1_22.

6 Luís Caires, Frank Pfenning, and Bernardo Toninho. Linear logic propositions as session types. *Math. Struct. Comput. Sci.*, 26(3):367–423, 2016. doi:10.1017/S0960129514000218.

7 Luca Ciccone and Luca Padovani. An infinitary proof theory of linear logic ensuring fair termination in the linear π-calculus. In Bartek Klin, Slawomir Lasota, and Anca Muscholl, editors, *33rd International Conference on Concurrency Theory, CONCUR 2022, September 12-16, 2022, Warsaw, Poland*, volume 243 of *LIPIcs*, pages 36:1–36:18. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CONCUR.2022.36.

8 Amina Doumane. *On the infinitary proof theory of logics with fixed points. (Théorie de la démonstration infinitaire pour les logiques à points fixes)*. PhD thesis, Paris Diderot University, France, 2017. URL: https://tel.archives-ouvertes.fr/tel-01676953.

9 Thomas Ehrhard. An introduction to differential linear logic: proof-nets, models and antiderivatives. *Math. Struct. Comput. Sci.*, 28(7):995–1060, 2018. doi:10.1017/S0960129516000372.

10 Nissim Francez. *Fairness.* Texts and Monographs in Computer Science. Springer, 1986. doi:10.1007/978-1-4612-4886-6.

11 Jean-Yves Girard and Yves Lafont. Linear logic and lazy computation. In Hartmut Ehrig, Robert A. Kowalski, Giorgio Levi, and Ugo Montanari, editors, *TAPSOFT'87: Proceedings of the International Joint Conference on Theory and Practice of Software Development, Pisa, Italy, March 23-27, 1987, Volume 2: Advanced Seminar on Foundations of Innovative Software Development II and Colloquium on Functional and Logic Programming and Specifications (CFLP)*, volume 250 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 1987. doi:10.1007/BFb0014972.

12 Jean-Yves Girard, Andre Scedrov, and Philip J. Scott. Bounded linear logic: A modular approach to polynomial-time computability. *Theor. Comput. Sci.*, 97(1):1–66, 1992. doi:10.1016/0304-3975(92)90386-T.

13 Orna Grumberg, Nissim Francez, and Shmuel Katz. Fair termination of communicating processes. In *Proceedings of the Third Annual ACM Symposium on Principles of Distributed Computing*, PODC '84, pages 254–265, New York, NY, USA, 1984. Association for Computing Machinery. doi:10.1145/800222.806752.

14 Kohei Honda. Types for dyadic interaction. In Eike Best, editor, *CONCUR '93, 4th International Conference on Concurrency Theory, Hildesheim, Germany, August 23-26, 1993, Proceedings*, volume 715 of *Lecture Notes in Computer Science*, pages 509–523. Springer, 1993. doi:10.1007/3-540-57208-2_35.

15 Kohei Honda, Vasco Thudichum Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In Chris Hankin, editor, *Programming Languages and Systems - ESOP'98, 7th European Symposium on Programming, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'98, Lisbon, Portugal, March 28 - April 4, 1998, Proceedings*, volume 1381 of *Lecture Notes in Computer Science*, pages 122–138. Springer, 1998. doi:10.1007/BFb0053567.

16 Hans Hüttel, Ivan Lanese, Vasco T. Vasconcelos, Luís Caires, Marco Carbone, Pierre-Malo Deniélou, Dimitris Mostrous, Luca Padovani, António Ravara, Emilio Tuosto, Hugo Torres Vieira, and Gianluigi Zavattaro. Foundations of session types and behavioural contracts. *ACM Comput. Surv.*, 49(1):3:1–3:36, 2016. doi:10.1145/2873052.

17 Wen Kokke, J. Garrett Morris, and Philip Wadler. Towards races in linear logic. *Log. Methods Comput. Sci.*, 16(4), 2020. URL: https://lmcs.episciences.org/6979.

18 Leslie Lamport. Fairness and hyperfairness. *Distributed Comput.*, 13(4):239–245, 2000. doi:10.1007/PL00008921.

19 Sam Lindley and J. Garrett Morris. Talking bananas: structural recursion for session types. In Jacques Garrigue, Gabriele Keller, and Eijiro Sumii, editors, *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming, ICFP 2016, Nara, Japan, September 18-22, 2016*, pages 434–447. ACM, 2016. doi:10.1145/2951913.2951921.

20 Zesen Qian, G. A. Kavvos, and Lars Birkedal. Client-server sessions in linear logic. *Proc. ACM Program. Lang.*, 5(ICFP):1–31, 2021. doi:10.1145/3473567.

21 Philip Wadler. Propositions as sessions. *J. Funct. Program.*, 24(2-3):384–418, 2014. doi:10.1017/S095679681400001X.

A Supplement to Section 3

In the proofs of Lemmas 18 and 19 below we only focus on the *derivability* of the typing judgment a structural pre-congruence or a reduction, without worrying about the *validity* of the derivation. It is easy to see that validity is preserved since both structural pre-congruence and reductions either change a *finite* region of the typing derivation or *remove* an entire sub-tree of the derivation (as in the case of [R-CASE]). Either way, the fact that every infinite branch in the residual derivation satisfies the validity conditions (Definition 6) follows from the hypothesis that the initial typing derivation is valid.

▶ **Lemma 18.** *If $P \vdash \Gamma$ and $P \preccurlyeq Q$ then $Q \vdash \Gamma$.*

Proof. By induction on the derivation of $P \preccurlyeq Q$ and by cases on the last rule applied. The proof is standard, we only discuss [s-par-pool] for illustration purposes. In this case $P = (z)(¿x[y].P_1 :: P_2 \mid P_3) \preccurlyeq ¿x[y].P_1 :: (z)(P_2 \mid P_3) = Q$ where $z \not\in \mathsf{fn}(¿x[y].P_1)$. From [cut] we deduce $¿x[y].P_1 :: P_2 \vdash \Gamma_{12}, z : T$ and $P_3 \vdash \Gamma_3, z : T^\perp$ where $\Gamma = \Gamma_{12}, \Gamma_3$. From [client] and $z \not\in \mathsf{fn}(¿x[y].P_1)$ we deduce $P_1 \vdash \Gamma_1, y : S$ and $P_2 \vdash \Gamma_2, x : ¿S, z : T$ where $\Gamma_{12} = \Gamma_1, \Gamma_2, x : ¿S$. We derive $(z)(P_2 \mid P_3) \vdash \Gamma_2, \Gamma_3, x : ¿S$ with one application of [cut]. We conclude $Q \vdash \Gamma$ with one application of [client]. ◀

▶ **Lemma 19.** *If $P \vdash \Gamma$ and $P \to Q$ then $Q \vdash \Gamma$.*

Proof. By induction on the derivation of $P \to Q$ and by cases on the last rule applied.

- [r-close] Then $P = (x)(\mathsf{close}\, x \mid \mathsf{wait}\, x.Q) \to Q$. From [cut], [$\mathbf{1}$] and [$\perp$] we deduce $\mathsf{close}\, x \vdash x : \mathbf{1}$ and $\mathsf{wait}\, x.Q \vdash \Gamma, x : \perp$. From [$\perp$] we conclude $Q \vdash \Gamma$.
- [r-comm] Then $P = (x)(x[y](P_1 \mid P_2) \mid x(y).P_3) \to (y)(P_1 \mid (x)(P_2 \mid P_3)) = Q$. From [cut] we deduce $x[y](P_1 \mid P_2) \vdash \Gamma_{12}, x : T \otimes S$ and $x(y).P_3 \vdash \Gamma_3, x : T^\perp \parr S^\perp$ where $\Gamma = \Gamma_{12}, \Gamma_3$. From [$\otimes$] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : S$ where $\Gamma_{12} = \Gamma_1, \Gamma_2$. From [$\parr$] we deduce $P_3 \vdash \Gamma_3, y : T^\perp, x : S^\perp$. We derive $(x)(P_2 \mid P_3) \vdash \Gamma_2, \Gamma_3, y : T^\perp$ with one application of [cut]. We conclude $(y)(P_1 \mid (x)(P_2 \mid P_3)) \vdash \Gamma$ with one application of [cut].
- [r-case] Then $P = (x)(\mathsf{in}_i\, x.R \mid \mathsf{case}\, x\{Q_1, Q_2\}) \to (x)(R \mid Q_i) = Q$. From [cut], [$\oplus$] and [$\&$] we deduce $\mathsf{in}_i\, x.R \vdash \Gamma_1, x : T_1 \oplus T_2$ and $\mathsf{case}\, x\{Q_1, Q_2\} \vdash \Gamma_2, x : T_1^\perp \mathbin{\&} T_2^\perp$ where $\Gamma = \Gamma_1, \Gamma_2$. From [$\oplus$] we deduce $R \vdash \Gamma_1, x : T_i$. From [$\&$] we deduce $Q_i \vdash \Gamma_2, x : T_i^\perp$ for $i = 1, 2$. We conclude $(x)(R \mid Q_i) \vdash \Gamma$ with one application of [cut].
- [r-connect] Then $P = (x)(¿x[y].P_1 :: P_2 \mid ¡x(y)\{Q_1, Q_2\}) \to (y)(P_1 \mid (x)(P_2 \mid Q_1)) = Q$. From [cut], [client] and [server] we deduce $¿x[y].P_1 :: P_2 \vdash \Gamma_{12}, x : ¿T$ and $¡x(y)\{Q_1, Q_2\} \vdash \Delta, x : ¡T^\perp$ where $\Gamma = \Gamma_{12}, \Delta$. From [client] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : ¿T$ where $\Gamma_{12} = \Gamma_1, \Gamma_2$. From [server] we deduce $Q_1 \vdash \Delta, x : ¡T^\perp, y : T^\perp$. We derive $(x)(P_2 \mid Q_1) \vdash \Gamma_2, \Delta, y : T^\perp$ with one application of [cut]. We conclude $(y)(P_1 \mid (x)(P_2 \mid Q_1)) \vdash \Gamma$ with another application of [cut].
- [r-done] Then $P = (x)(¿x[] \mid ¡x(y)\{R, Q\}) \to Q$. From [cut], [done] and [server] we deduce $¿x[] \vdash x : ¿T$ and $¡x(y)\{R, Q\} \vdash \Gamma, x : ¡T^\perp$. From [server] we conclude $Q \vdash \Gamma$.
- [r-par] Then $P = (x)(P_1 \mid P_2) \to (x)(Q_1 \mid P_2) = Q$ where $P_1 \to Q_1$. From [cut] we deduce $P_1 \vdash \Gamma_1, x : T$ and $P_2 \vdash \Gamma_2, x : T^\perp$ where $\Gamma = \Gamma_1, \Gamma_2$. Using the induction hypothesis we derive $Q_1 \vdash \Gamma_1, x : T$. We conclude $(x)(Q_1 \mid P_2) \vdash \Gamma$ with an application of [cut].
- [r-pool] Then $P = ¿x[y].P_1 :: P_2 \to ¿x[y].P_1 :: Q_2 = Q$ where $P_1 \to Q_2$. From [client] we deduce $P_1 \vdash \Gamma_1, y : T$ and $P_2 \vdash \Gamma_2, x : ¿T$ where $\Gamma = \Gamma_1, \Gamma_2, x : ¿T$. Using the induction hypothesis we derive $Q_2 \vdash \Gamma_2, x : ¿T$. We conclude $¿x[y].P_1 :: Q_2 \vdash \Gamma$ with an application of [client].
- [r-struct] Using the induction hypothesis with two applications of Lemma 18. ◀

In order to prove deadlock freedom it is convenient to introduce *reduction contexts* to make it easy to refer to unguarded sub-terms of a process. A reduction context is basically a process with a single hole denoted by $[\,]$.

$$\textbf{Reduction context} \qquad \mathcal{C}, \mathcal{D} \quad ::= \quad [\,] \quad \mid \quad (x)(\mathcal{C} \mid P) \quad \mid \quad (x)(P \mid \mathcal{C})$$

Note that holes cannot occur in the tail of client pools, that is $¿x[y].P :: \mathcal{C}$ is *not* a reduction context even though the tail of a client pool may reduce by means of [r-pool]. The point is that, in order to prove deadlock freedom, it is never necessary to reduce the tail of a client pool. Hereafter we write $\mathcal{C}[P]$ for the process obtained by replacing the hole in $\mathcal{C}$ with P. Note that this notion of replacement may capture some channels occurring free in P.

Before addressing deadlock freedom, we prove the following proximity lemma, showing that it is always possible to move a restriction close to a process in which the restricted channel occurs free.

▶ **Lemma 20.** *If* $x \in \mathsf{fn}(P) \setminus (\mathsf{fn}(\mathcal{C}) \cup \mathsf{bn}(\mathcal{C}))$ *then* $(x)(\mathcal{C}[P] \mid Q) \preccurlyeq \mathcal{D}[(x)(P \mid Q)]$ *for some* $\mathcal{D}$*.*

Proof. By induction on $\mathcal{C}$ and by cases on its shape. We do not detail symmetric cases and we assume, without loss of generality, that $\mathsf{fn}(Q) \cap \mathsf{bn}(\mathcal{C}) = \emptyset$.

- $\mathcal{C} = [\,]$. We conclude by taking $\mathcal{D} \stackrel{\mathsf{def}}{=} [\,]$ and by reflexivity of $\preccurlyeq$.
- $\mathcal{C} = (y)(\mathcal{C}' \mid R)$. Then $x \in \mathsf{fn}(P) \setminus (\mathsf{fn}(\mathcal{C}') \cup \mathsf{bn}(\mathcal{C}') \cup \mathsf{fn}(R) \cup \{y\})$. We derive

$$\begin{array}{rll}
(x)(\mathcal{C}[P] \mid Q) & = (x)((y)(\mathcal{C}'[P] \mid R) \mid Q) & \text{by definition of } \mathcal{C} \\
& \preccurlyeq (x)(Q \mid (y)(\mathcal{C}'[P] \mid R)) & \text{by [\textsc{s-par-comm}]} \\
& \preccurlyeq (y)((x)(Q \mid \mathcal{C}'[P]) \mid R) & \text{by [\textsc{s-par-assoc}] since } x \notin \mathsf{fn}(R),\ y \notin \mathsf{fn}(Q) \\
& \preccurlyeq (y)((x)(\mathcal{C}'[P] \mid Q) \mid R) & \text{by [\textsc{s-par-comm}]} \\
& \preccurlyeq (y)(\mathcal{D}'[(x)(P \mid Q)] \mid R) & \text{by ind. hyp. for some } \mathcal{D}' \\
& = \mathcal{D}[(x)(P \mid Q)] & \text{by taking } \mathcal{D} \stackrel{\mathsf{def}}{=} (y)(\mathcal{D}' \mid R)
\end{array}$$

◀

The next auxiliary result proves that, in a well-typed process, a finite number of applications of [S-CALL] is always sufficient to unfold all of the process invocations occurring in it. To this aim, we introduce some more terminology on processes. We say that P is a *guard* if it is not a parallel composition or a process invocation. Note that every guard specifies a topmost action on some channel x. In this case, we say that P is an x-guard. We say that P is *unguarded* in Q if $Q = \mathcal{C}[P]$ for some $\mathcal{C}$. We say that P is *unfolded* if $P = \mathcal{C}[Q]$ implies that Q is not an invocation.

▶ **Lemma 21.** *If* $P \vdash \Gamma$ *then there exists an unfolded* Q *such that* $P \preccurlyeq Q$*.*

Proof. Let the *call depth* of P be the natural number $\mathsf{cd}(P)$ inductively defined as follows:

$$\mathsf{cd}(P) = \begin{cases} 1 + \mathsf{cd}(Q) & \text{if } P = \mathsf{A}\langle \overline{x} \rangle \text{ and } \mathsf{A}(\overline{x}) \triangleq Q \\ 1 + \max\{\mathsf{cd}(P_1), \mathsf{cd}(P_2)\} & \text{if } P = (x)(P_1 \mid P_2) \\ 0 & \text{otherwise} \end{cases}$$

Roughly, $\mathsf{cd}(P)$ is the maximum depth in the typing derivation of P where an unguarded guard is encountered. To see that $\mathsf{cd}(P)$ is well defined, recall that in every infinite branch of a valid typing derivation there are infinitely many applications of [SERVER] and that a process of the form $¡x(y)\{Q, R\}$ is a guard. Therefore, the value of $\mathsf{cd}(P)$ is only determined by the portion of P's derivation tree that stops at each occurrence of a guard. This portion is finite. The proof proceeds by induction on $\mathsf{cd}(P)$ and by cases on the shape of P. The desired Q is obtained by applying [S-CALL] each time an unguarded invocation is encountered and the induction guarantees that this rewriting is finite. ◀

▶ **Theorem 9** (deadlock freedom). *If* $P \vdash x : \mathbf{1}$ *then either* $P \preccurlyeq \mathsf{close}\, x$ *or* $P \rightarrow_{\mathsf{det}}$*.*

Proof. By Lemma 21 we may assume, without loss of generality, that P is unfolded. We want to show that there are two x-guards in P that can synchronize. To this aim, let $\mathsf{guards}(P)$ be inductively defined as

$$\mathsf{guards}(P) = \begin{cases} \mathsf{guards}(P_1) + \mathsf{guards}(P_2) & \text{if } P = (x)(P_1 \mid P_2) \\ 1 & \text{otherwise} \end{cases}$$

and let $\mathsf{channels}(P)$ be inductively defined as

$$\mathsf{channels}(P) = \begin{cases} 1 + \mathsf{channels}(P_1) + \mathsf{channels}(P_2) & \text{if } P = (x)(P_1 \mid P_2) \\ 0 & \text{otherwise} \end{cases}$$

In words, $\mathsf{guards}(P)$ counts the number of unguarded guards in P whereas $\mathsf{channels}(P)$ counts the number of unguarded restrictions in P. It is easy to prove that $\mathsf{guards}(P) > \mathsf{channels}(P)$. So, there must be at least one channel name x such that P contains two unguarded x-guards. That is, $P = \mathcal{C}[(x)(\mathcal{C}_1[P_1] \mid \mathcal{C}_2[P_2])]$ and both P_1 and P_2 are x-guards and $x \not\in \mathsf{fn}(\mathcal{C}_i) \cup \mathsf{bn}(\mathcal{C}_i)$ for $i = 1, 2$. Then we derive

$$\begin{array}{rcll} P & = & \mathcal{C}[(x)(\mathcal{C}_1[P_1] \mid \mathcal{C}_2[P_2])] & \text{by definition of } P,\, P_1 \text{ and } P_2 \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[(x)(P_1 \mid \mathcal{C}_2[P_2])]] & \text{by Lemma 20 for some } \mathcal{D}_1 \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[(x)(\mathcal{C}_2[P_2] \mid P_1)]] & \text{by } [\textsc{s-par-comm}] \\ & \preccurlyeq & \mathcal{C}[\mathcal{D}_1[\mathcal{D}_2[(x)(P_2 \mid P_1)]]] & \text{by Lemma 20 for some } \mathcal{D}_2 \end{array}$$

Now we reason by cases on the shape of P_1 and P_2, knowing that they are x-guards and that they are well typed in contexts that contain the associations $x : T$ and $x : T^\perp$ for some T. If $P_2 = ¿x[y].Q$ then $P_1 = ¡x(y)\{Q_1, Q_2\}$ and P may reduce using [R-CONNECT]. The case in which $P_1 = ¿x[y].Q$ is symmetric and can be handled in a similar way with an additional application of [S-PAR-COMM]. The cases in which one of P_1 and P_2 is $¿x[]$ can be handled analogously, deducing that P may reduce using [R-DONE]. The only cases left are when neither P_1 nor P_2 is a client or $¿x[]$. Then, P_1 and P_2 must be x-guards beginning with dual actions which can synchronize using one of the rules [R-CLOSE], [R-COMM] or [R-CASE], possibly with the help of an application of [S-PAR-COMM]. ◀

`mitten`: A Flexible Multimodal Proof Assistant

Philipp Stassen
Aarhus University, Denmark

Daniel Gratzer
Aarhus University, Denmark

Lars Birkedal
Aarhus University, Denmark

Abstract

Recently, there has been a growing interest in type theories which include *modalities*, unary type constructors which need not commute with substitution. Here we focus on MTT [15], a general modal type theory which can internalize arbitrary collections of (dependent) right adjoints [7]. These modalities are specified by mode theories [20], 2-categories whose objects corresponds to modes, morphisms to modalities, and 2-cells to natural transformations between modalities. We contribute a defunctionalized NbE algorithm which reduces the type-checking problem for MTT to deciding the word problem for the mode theory. The algorithm is restricted to the class of *preordered* mode theories – mode theories with at most one 2-cell between any pair of modalities. Crucially, the normalization algorithm does not depend on the particulars of the mode theory and can be applied without change to any preordered collection of modalities. Furthermore, we specify a bidirectional syntax for MTT together with a type-checking algorithm. We further contribute `mitten`, a flexible experimental proof assistant implementing these algorithms which supports all decidable preordered mode theories without alteration.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Modal and temporal logics

Keywords and phrases Dependent type theory, guarded recursion, modal type theory, proof assistants

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.6

Funding This work was supported in part by a Villum Investigator grant (no. 25804), Center for Basic Research in Program Verification (CPV), from the VILLUM Foundation.

1 Introduction

A fundamental benefit of using type theory is the possibility of working within a proof assistant, which can check and even aid in the construction of complex theorems. Implementing a proof assistant, however, is a highly nontrivial task. In addition to a solid theoretical foundation for the particular type theory, numerous practical implementation issues must be addressed.

Recently, interest has gathered around type theories with *modalities*, unary type constructors which need not commute with substitution. Unfortunately, the situation for modal type theories is even more fraught; the theory for modalities is poorly understood in general, and it is unknown whether standard implementation techniques extend to support them.

Despite these challenges, mainstream proof assistants have begun to experiment with modalities [27], but these implementations are costly and only apply to a particular modal type theory. In practice, a type theorist may use a particular collection of modalities for only one proof or construction and it is impractical to invest in a specialized modal proof assistant each time. This churn has pushed type theorists to define *general* modal type theories which can be instantiated to a variety of modal situations [21, 16].

We choose to focus on MTT [15], a general modal type theory which can internalize an arbitrary collection of modalities so long as they behave like *right adjoints* [7]. Despite limiting consideration to right adjoints, MTT can be used to model a variety of existing

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 6; pp. 6:1–6:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

modal type theories including calculi for guarded recursion, internalized parametricity, and axiomatic cohesion. Better still, MTT has a robustly developed metatheory [15, 13] which applies *irrespective* of the chosen modalities. An implementation of MTT could therefore conceivably be designed to allow the user to freely change the collection of modalities without re-implementing the entire proof assistant each time. This, in turn, enables the kind of specialized modal proof assistants previously impractical for one-off modal type theories.

1.1 MTT: a general modal type theory

As mentioned, MTT can be instantiated with a collection of modalities. More precisely, MTT is parameterized by a mode theory, a strict 2-category which describes a modal situation. Intuitively, objects (m, n, o) of this mode theory represent distinct type theories which are then connected by 1-cells (μ, ν, ξ) which describe the modalities. The categorical structure ensures that modalities compose and that there is an identity modality. In order to describe more intricate connections and structure, the mode theory also contains 2-cells (α, β). A 2-cell induces a "natural transformation" between modalities. By carefully choosing 2-cells we can force a modality to e.g. become a comonad, a monad, or an adjoint.

To give a paradigmatic example, consider the mode theory $\mathcal{M}$ with a single object m, a single non-identity morphism $\mu : m \longrightarrow m$ and a 2-cell $\epsilon : \mu \longrightarrow \mathsf{id}_m$ subject to the equations $\mu \circ \mu = \mu$ and $\epsilon \star \mu = \mu \star \epsilon$. This description defines $\mathcal{M}$ as a 2-category with a strictly idempotent comonad μ. Instantiating MTT with this mode theory yields a modality $\langle \mu \mid - \rangle$ together with definable operations shaping $\langle \mu \mid - \rangle$ into an idempotent comonad:

$$\mathsf{extract}_A : \langle \mu \mid A \rangle \to A \qquad\qquad \mathsf{dup}_A : \langle \mu \mid A \rangle \simeq \langle \mu \mid \langle \mu \mid A \rangle \rangle$$

Even this simple modal type theory is quite useful; it can serve as a replacement for the experimental version of Agda [27] used to formalize a construction of univalent universes [19].

Given the generality, it is natural to wonder whether instantiating MTT yields a calculus which is feasible to work with in practice. Fortunately, prior Fitch-style type theories have been highly workable [4, 5, 26] and this trend has continued with MTT [15, 13, 14].

1.2 From theory to practice

Unfortunately, converting the theoretical guarantee of normalization into an executable program is not a small step. A first obstacle is the syntax of MTT itself: prior work has exclusively considered an algebraic presentation of the syntax as a generalized algebraic theory. While mathematically elegant, a proof assistant requires a more streamlined and ergonomic syntax. Once a more convenient syntax has been designed, one must adapt the normalization proof to a normalization algorithm. Normalization is proven by a sophisticated *gluing* argument, and while the proof is reminiscent of normalization-by-evaluation [2] it remains to extract such an algorithm. Finally, the normalization algorithm does not give any insight into representing common mode theories or solving their word problems.

Restriction to preordered mode theories

Many difficulties flow not from the modalities per se, but from the 2-cells of our mode theory, which induce a new primitive type of substitutions. During normalization these *key substitutions* accumulate at variables. Unfortunately, they disrupt a crucial property of modern NbE algorithms: variables can no longer be presented in a way that is invariant under weakening. Therefore, we restrict our attention to mode theories that are *preordered*, with at most one 2-cell between any pair of modalities.

This allows us to present a syntax that never talks about 2-cells and relies entirely on the elaboration procedure to insert and check 2-cells. In addition to avoiding annotations, this simplifies the normalization algorithm since the troublesome key substitutions trivialize.

Although such a restriction does preclude some examples, preordered mode theories are still expressive enough to model guarded recursion together with an *everything now* modality similar to the one introduced by Clousten et al. [10].

A surface syntax for MTT

As a generalized algebraic theory, MTT is presented with explicit substitutions and fully annotated connectives [15]. In order to avoid this bureaucracy, we introduce a bidirectional version of MTT which allows a user of `mitten` to omit almost all type annotations [11].

Normalization-by-evaluation

The normalization proof for MTT follows the structure of a normalization-by-evaluation proof. Rather than fixing a rewriting system, a term is *evaluated* into a carefully chosen semantics equipped with a quotation function reifying an element of the semantic domain to a normal form. The entire normalization function is then a round-trip from syntax to semantics and then back to normal forms. While the proof of normalization uses a traditional denotational model for a semantic domain, this approach is unsuitable for implementation.

Instead `mitten` follows the literature on normalization-by-evaluation and uses a *defunctionalized* and *syntactic* semantic domain [2]. This approach has previously been adapted to work with particular a modal type theories [17, 18].

Mode theories

As mentioned previously, normalization for MTT does not immediately imply the decidability of type equality. Terms (and therefore types) mention both 1- and 2-cells from the mode theory, and deciding the mode theory is a necessary precondition for deciding type equality. Moreover, deciding the equality of 1- and 2-cells, even in a finitely presented 2-category, is well-known to be undecidable.[1] For us, this situation is slightly improved since for preordered mode theories at least 2-cell equality is trivial. Unfortunately, the undecidability of 1-cell equality remains. Special attention is therefore necessary for each mode theory to ensure that the normalization algorithm for MTT is sufficient to yield a type-checker.

While this rules out a truly generic proof assistant for MTT which works regardless of the choice of mode theory, `mitten` shows that the best theoretically possible result is obtainable. We implement `mitten` to be parameterized by a module describing the mode theory so that the type-checker relies only on the existence of such a decision procedure. In particular, there is no need to alter the entire proof assistant when changing the mode theory; only a new mode theory module is necessary. Crucially, while the user must write a small amount of code, no specialized knowledge of proof assistants is required.

We have implemented several mode theories commonly used with MTT in this way, showing that in practice decidability is no real obstacle. For instance, we have configured `mitten` to support guarded recursion with a combination of two modalities $\Box$ and $\blacktriangleright$. This is the first proof assistant to support this combination of modalities.

[1] The word problem is well-known to be undecidable for finitely presented groups which can be realized as finitely-presented categories and therefore locally discrete finitely-presentable 2-categories.

1.3 Contributions

We contribute a bidirectional syntax for MTT (restricted to preordered mode theories) together with a defunctionalized normalization-by-evaluation algorithm which reduces the type-checking problem to deciding the word problem for the mode theory. We have put these results into practice with `mitten`, a prototype implementation of MTT based on this algorithm. `mitten` also supports the replacement of the underlying mode theory with minimal alterations, allowing a user to construct specialized proof assistants for modal type theories by merely supplying a single module specifying the mode theory together with equality functions for 0-, 1-, and 2-cells.

In Section 2 we provide a guided tour of MTT. This section also introduces the bidirectional syntax for MTT and shows how even in this general setting the modalities introduce minimal overhead. Section 3 introduces the defunctionalized normalization algorithm for non-specialists and Sections 4 and 5 completes the description of the core components of `mitten` by describing the type-checking algorithm. In so doing, we also describe the novel interface `mitten` uses to represent modalities and show how this interface is implemented.

In Section 6 we discuss the realization of mode theories with a representative example: guarded recursion. As previously mentioned, this is the first proof assistant able to support this pair of modalities simultaneously.

2 A surface syntax for MTT

Prior to specifying a type-checking algorithm for MTT, we must specify the surface syntax for the language. This question is not satisfactorily addressed in the prior work on MTT; the *generalized algebraic* version of syntax is too verbose to be workable, but the informal pen-and-paper syntax which omits all type annotations cannot be type-checked. Our surface syntax is formulated with an eye towards the type-checking algorithm we will eventually use: a version of Coquand's semantic type-checker [11]. In particular, we will employ a bidirectional surface syntax which minimizes the number of mandatory annotations while still ensuring the decidability of type-checking.

To a first approximation, the surface syntax is divided into two components: checkable and synthesizable terms. Checkable terms include introduction forms while synthesizable terms include elimination forms and variables. By carefully controlling where checkable and synthesizable terms are used, we thereby avoid unnecessary type annotations.

We present the grammar for surface syntax in Section 2.1. While we will defer presenting the actual type-checking algorithm until Section 5, in order to make this account as self-contained as possible we provide an example-based introduction to MTT in Section 2.2.

2.1 Bidirectional Syntax

As previously mentioned, MTT is parameterized by a mode theory [20] which specifies the modes and modalities of the type theory. We begin by more precisely defining a mode theory in our situation.

▶ **Definition 1.** *A mode theory is a category whose objects m, n, o we refer to as* modes *and whose morphisms μ, ν we refer to as* modalities*. We further require that each hom-set be equipped with a pre-order $\leq$ compatible with composition. Explicitly, given $\mu, \nu \in \mathsf{Hom}(m, n)$ and $\rho, \sigma \in \mathsf{Hom}(n, o)$ with $\mu \leq \nu$ and $\rho \leq \sigma$ we require $\rho \circ \mu \leq \sigma \circ \nu$.*

Equivalently, a mode theory is a preorder-enriched category.

For the remainder of this subsection, we fix a mode theory $\mathcal{M}$. The grammar of the surface syntax is presented below:

$$\begin{array}{lll} \textit{(Checkable)} & A, M, N, C & ::= & R \mid (\mu \mid A) \to B \mid A{\times}B \mid \texttt{Nat} \mid \texttt{Id}_A(M,N) \mid \texttt{U} \mid \langle \mu \mid A \rangle \\ & & & \mid (M,N) \mid \texttt{zero} \mid \texttt{succ}(M) \mid \lambda(M) \mid \texttt{mod}_\mu(M) \mid \texttt{refl}_M \\ \textit{(Synthesizable)} & R, S & ::= & M : A \mid \mathbf{q}_k \mid R(M)_\mu \mid \texttt{pr}_1(R) \mid \texttt{pr}_2(R) \\ & & & \mid \texttt{let}_\mu\ \texttt{mod}_\nu(\_) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ C \mid \texttt{rec}(C, M_{\mathsf{zero}}, M_{\mathsf{suc}}, N) \\ & & & \mid \texttt{J}(C, c_{\mathsf{refl}}, M) \end{array}$$

As mentioned previously, checkable terms consist essentially of introduction forms while synthesizable terms are elimination principles. For instance, the presentation of dependent sums above includes $A{\times}B$ and (M,N) as checkable terms while $\texttt{pr}_i(R)$ is synthesizable.

By stratifying terms in this way we ensure that annotations are required exactly where ambiguity would arise during type-checking. For instance, this stratification prevents unannotated β-redexes from occurring. Consider again the case of dependent sums. In order to apply a projection to an element (M,N) of dependent sum type, the element must be synthesizable. However, since (M,N) is checkable, the only way to represent $\texttt{pr}_1((M,N))$ in this discipline is to promote (M,N) to a synthesizable term by annotating it: $(M,N) : A{\times}B$.

▶ Remark 2. In particular, terms in β-normal and η-long form fit into this surface syntax with no additional annotations. Consequently, the normalization theorem for MTT [13] ensures that any term is convertible to one expressible in the surface syntax.

▶ Remark 3. We have made a concession to simplicity and used de Bruijn indices for variables rather than names. This makes the normalization and type-checking algorithms far easier to specify and it is well-known how to pass between syntax with named variables and de Bruijn indices. We will use named variables in examples e.g., $\texttt{let}_\mu\ \texttt{mod}_\nu(y) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ x.C$ or $(\mu \mid x : A) \to B$ for modal elimination and dependent products respectively.

2.2 The surface syntax by example

We will crystallize when a term in the surface syntax is well-formed in Section 5 when presenting the type-checking algorithm. In order to cultivate intuition for the theory before this, we will now work through several examples in the language.

▶ Remark 4. We refer the reader to Gratzer et al. [15] for a long form explanation of MTT.

MTT with one mode and one generating modality

Consider MTT instantiated with the mode theory with one mode m and one modality ϕ with no non-trivial equations or inequalities. Then each modality μ is uniquely expressible as ϕ^n, the composition of n copies of ϕ. Just as in ordinary type theory, MTT then has dependent sums, natural numbers, identity types, and their behavior is unchanged.

Unlike in ordinary type theory, each variable is annotated with a modality $x : (\mu \mid A)$ (pronounced $x : A$ annotated by μ). Variables annotated with the identity modality behave like 'ordinary' variables; they can be used freely when working with e.g. natural numbers. Conversely, variables annotated with ϕ^{n+1} cannot be used except in the construction of an element the modal type $\langle \phi \mid A \rangle$.

An element of $\langle \phi \mid A \rangle$ is introduced by $\texttt{mod}_\phi(M)$, where M is an element of A, subject to the restriction that M may only use variables with annotation ϕ^{n+1}. More concretely, when we construct M we (1) lose access to all id-annotated variables and (2) replace a variable $x : (\phi^{n+1} \mid A)$ with $x : (\phi^n \mid A)$. As only variables with identity annotation can be used with the variable rule, this means that within $\texttt{mod}_\phi(-)$ we may use ϕ-annotated variables freely.

For instance, in the context with variables $x_0 : (\mathsf{id} \mid \texttt{Nat})$, $x_1 : (\phi \mid \texttt{Nat})$, and $x_2 : (\phi \circ \phi \mid \texttt{Nat})$ the following programs are well-typed:

$$x_0 : \mathsf{Nat} \qquad \texttt{mod}_{\phi}(x_1) : \langle \phi \mid \texttt{Nat} \rangle \qquad \texttt{mod}_{\phi}(\texttt{mod}_{\phi}(x_2)) : \langle \phi \mid \langle \phi \mid \texttt{Nat} \rangle \rangle$$

On the other hand, both $x_1 : \texttt{Nat}$ and $\texttt{mod}_{\phi}(x_0) : \langle \phi \mid \texttt{Nat} \rangle$ are ill-typed as the annotations on variables do not match their usage.

This idea generalizes: to construct an element of $\langle \phi^k \mid A \rangle$ we use $\texttt{mod}_{\phi^k}(M)$ where $M : A$ in a context where we have (1) lost access to variables with annotations ϕ^l where $l < k$ (2) replaced each variable $x : (\phi^{n+k} \mid A)$ with $x : (\phi^n \mid A)$. In the same context as the example above therefore, $\texttt{mod}_{\phi \circ \phi}(x_2) : \langle \phi \circ \phi \mid \texttt{Nat} \rangle$. We refer to the modification to the context given by (1) and (2) as *ϕ^k-restricting the context.*

Let us now consider the modal function type $(\mu \mid A) \to B$. An element of $(\mu \mid A) \to B$ is precisely a function which binds a variable of type A with annotation μ. Application for these function types $R(M)_\mu$ takes μ into account in the following way: $R(M)_\mu : B$ if (1) R has type $(\mu \mid A) \to B$ and (2) after μ-restricting the context, M has type A.

One feature remains to be discussed, the elimination principle for modal types:

$$\texttt{let}_{\nu}\ \texttt{mod}_{\mu}(y) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ x.\,C$$

To a first approximation, this principle allows us to replace a variable $x : (\nu \mid \langle \mu \mid A \rangle)$ with $y : (\nu \circ \mu \mid A)$. More precisely, $\texttt{let}_{\nu}\ \texttt{mod}_{\mu}(y) \leftarrow R\ \texttt{in}\ M\ \texttt{over}\ x.C : C[M/x]$ if (1) after binding $x : (\nu \mid \langle \mu \mid A \rangle)$, C is a type (2) after ν-restricting the context M has type $\langle \mu \mid A \rangle$ and (3) after binding $y : (\nu \circ \mu \mid A)$, R has type $C[\texttt{mod}_{\mu}(y)/x]$.

Multiple modalities

The above approach for ϕ-restriction based on decrementing modal annotations provides a simple mental model for MTT. To extend these ideas to more complex mode theories, however, a more refined approach is necessary. We begin by discussing a small adjustment to the concepts introduced previously.

Rather than eagerly decrementing the annotation on a variable when we restrict a context, we instead *lazily* perform this update. Accordingly, we annotate each variable with a pair of modalities and write $x :_{\mu/\nu} A$ for a μ-annotated variable with a ν-restriction lazily performed upon it. The rule for applying a restriction to a variable now becomes more uniform: to restrict $x :_{\mu/\nu} A$ by ξ we replace it with $x :_{\mu/\nu\circ\xi} A$. The variable rule applies only when the fraction "cancels" i.e., $x :_{\mu/\mu} A \vdash x : A$.

For the mode theory under consideration, this is merely a change in notation as the behavior of the annotations of $x :_{\phi^l/\phi^k} A \vdash x : A$ is entirely determined by the difference $l - k$. We therefore introduce the following mode theory to illustrate the need for the "lazy" approach:

▶ **Definition 5.** *Denote by $\mathcal{M}_1^{\mathsf{ex}}$ the mode theory with one mode and two generating modalities ψ and ϕ. The preorder is generated by the inequality $\psi \circ \psi \leq \phi$.*

This mode theory introduces two new concepts simultaneously: multiple modalities and non-trivial inequalities between those modalities. Fortunately, to refine the idea explained above of μ-restricting a context, only one rule must be altered: To account for the preorder on modalities, we relax the variable rule slightly: $x :_{\mu/\nu} A \vdash x : A$ if $\mu \leq \nu$. With this modified rule, we can construct a coercion $\langle \psi \circ \psi \mid A \rangle \to \langle \phi \mid A \rangle$:

$$\mathsf{coerce} = \lambda x.\ \texttt{let}_{\mathsf{id}}\ \texttt{mod}_{\psi\circ\psi}(y) \leftarrow x\ \texttt{in}\ \texttt{mod}_{\phi}(y)\ \texttt{over}\ \_.\,\langle \phi \mid A \rangle$$

Multiple modes and multiple modalities

Only one generalization is required at this point to provide a complete description of MTT: multiple modes. While thus far we have confined ourselves to discussing multiple modalities on one mode, we are allowed to have multiple modes in MTT as well. Consider the following mode theory:

▶ **Definition 6.** *$\mathcal{M}_2^{\mathsf{ex}}$ is the mode theory equipped with two modes k and l whose modalities are generated by $\phi : k \to k$ and $\psi, \xi : k \to l$. The preorder on hom-sets are generated by the inequalities $\mathsf{id}_k \leq \phi$ and $\xi \leq \psi$:*

$$\begin{tikzcd} k \arrow[loop left, "\phi"] \arrow[r, bend left, "\psi"] \arrow[r, bend right, "\xi"'] & l \end{tikzcd}$$

We note that now $\mathcal{M}_2^{\mathsf{ex}}$ now has two different modes k and l. Each mode in MTT gives rise to a separate type theory so that we must check not only that some term has a type, but also that the term, type, and all variables in scope live at the correct mode.

All of the standard constructions do not change the mode; thus, e.g., $\mathtt{succ}(n)$ will be well-typed at type $\mathtt{Nat}$ at mode m just when the same is true of n. We will notate "M has type A at mode m" by $M : A \mathbin{@} m$. Prior to discussing the two type constructors involving modalities, we must explain what it means for a context to be well-formed at mode m.

▶ **Definition 7.** *A variable declaration $x :_{\mu/\nu} A$ is well-formed at mode m if the following hold:*

1. *$\mu : n \longrightarrow o$ and $\nu : m \longrightarrow o$ for some o.*
2. *A is a type at mode n.*

The context is well-formed at mode m if all variables in scope are well-formed at mode m.

▶ **Example 8.** Restricting a well-formed context at m by $\mu : n \longrightarrow m$ yields a well-formed context in mode n.

It is worth emphasizing the contravariant nature of the restriction ν in $x :_{\mu/\nu} A$. This is crucial for the rules governing $\langle \mu \mid A \rangle$. The type $\langle \mu \mid A \rangle$ is well-formed at mode m if (1) $\mu : n \longrightarrow m$ for some n and (2) after μ-restricting the context, A is well-formed at mode n. In particular, $\langle \mu \mid - \rangle$ sends types at mode n to types at mode m so restriction must move contexts contravariantly from mode n to mode m. We remark, however, that aside from the additional checks to ensure that types are well-moded, this is the same rule as given previously. Likewise, the rules for introduction and elimination along with all of those for modal dependent products are merely instrumented with additional checks to ensure that types and terms live at the correct mode.

We conclude with a few examples.

▶ **Example 9.** $\lambda x.x : (\xi \mid A) \to \langle \psi \mid A \rangle \mathbin{@} l$ is well typed. In particular, since $\xi \leq \psi$ we conclude $x :_{\xi/\psi} A \vdash x : A \mathbin{@} k$.

▶ **Example 10.** We will define a function of the following type:

$$f : \langle \psi \mid \langle \phi \mid \mathtt{Nat} \rangle \rangle \to \langle \psi \circ \phi \mid \mathtt{Nat} \rangle \mathbin{@} l$$

We begin by binding a variable $x :_{\mathsf{id}/\mathsf{id}} \langle \psi \mid \langle \phi \mid \mathtt{Nat} \rangle \rangle$ so it now suffices to construct a term $\langle \psi \circ \phi \mid \mathtt{Nat} \rangle \mathbin{@} l$. To this end, we use the modal elimination principle on x to obtain a new variable $y :_{\psi/\mathsf{id}} \langle \phi \mid \mathtt{Nat} \rangle$. Applying modal elimination to y, we obtain $z :_{\psi \circ \phi/\mathsf{id}} \mathtt{Nat}$.

We still wish to construct a term $\langle \psi \circ \phi \mid \texttt{Nat}\rangle$. Applying the modal introduction rule, we $\psi \circ \phi$ restrict the context (so y becomes $y :_{\psi\circ\phi/\psi\circ\phi} \texttt{Nat}$). Our goal is then Nat, so y suffices. All told, the term final term is as follows:

$$
\begin{array}{l}
\lambda x.\\
\quad \texttt{let}_{\mathsf{id}_k}\ \texttt{mod}_{\psi}(y) = x\ \texttt{in}\\
\quad\quad \texttt{let}_{\psi}\ \texttt{mod}_{\phi}(z) = y\ \texttt{in}\\
\quad\quad\quad \texttt{mod}_{\psi\circ\phi}(z)\\
\quad\quad \texttt{over}\ \langle \psi \circ \phi \mid \texttt{Nat}\rangle\\
\quad \texttt{over}\ \langle \psi \circ \phi \mid \texttt{Nat}\rangle
\end{array}
$$

3 Normalization by Evaluation

A crucial ingredient of any type checker is a procedure for determining when two types are equal. In mitten, we have implemented this decision procedure through a normalization algorithm: a function which sends a term to a corresponding *normal form*. The precise definition of normal form is then less important than the fact that definitional equality for normal forms is straightforward to decide. Writing NfTerms for the collection of normal forms, we view our normalization algorithm as a function:

$$\underline{\mathbf{norm}}_{\Gamma} : \mathsf{Syntax} \to \mathsf{NfTerms}$$

Merely having a function from syntax to normal forms, however, is insufficient to decide definitional equality. Accordingly, we are interested in normalization functions which satisfy the following properties:

▶ **Definition 11.** *A normalization function is called* complete *if* $\Gamma \vdash A = B \mathbin{@} m$ *implies* $\underline{\mathbf{norm}}_{\Gamma}(A) = \underline{\mathbf{norm}}_{\Gamma}(B)$

▶ **Definition 12.** *A normalization function is* sound *if* $\Gamma \vdash A \mathbin{@} m$ *implies* $\Gamma \vdash \underline{\mathbf{norm}}_{\Gamma}(A) = A \mathbin{@} m$.

Completeness states that normalization lifts to a function on syntax quotiented by definitional equality while soundness states that this induced function has a section. Taken together, therefore, we have the following:

▶ **Corollary 13.** *Let* $\underline{\mathbf{norm}}_{\Gamma}$ *be sound and complete then* $\Gamma \vdash A = B \mathbin{@} m$ *if and only if* $\underline{\mathbf{norm}}_{\Gamma}(A) = \underline{\mathbf{norm}}_{\Gamma}(B)$.

The traditional approach to constructing a normalization function is to specify an untyped rewriting system which directs and presents the equational theory. Equality of terms is then convertibility within this rewriting system so that strong normalization ensures both soundness and completeness. This approach, however, turns out to be unworkable for more elaborate dependent type theories with type-directed rules. One possible approach is to can refine a rewriting system to be type-directed system which – in conjunction with other mechanisms – can decide conversion directly [3], we adopt an entirely different approach to associating terms to normal forms: *normalization by evaluation* (NbE).

Normalization by evaluation breaks the process of normalizing a term into two distinct phases: evaluation and quotation. The first *evaluates* a term into a *semantic domain*. For our purposes, the semantic domain is simply a more restrictive form of syntax which disallows

β-reducible terms. The process of evaluation boils down to placing a term in β-normal form while crucially retaining various pieces of type information for the next phase. The second phase, quotation, takes an element of the semantic domain and *quotes* it back to syntax. In the process it η expands terms wherever possible. As a result, the full loop of evaluation and quotation sends a term to its β-normal η-long form as required. Figure 1 gives a graphical overview of the process.

We describe the semantic domain in detail in Section 3.1. The actual algorithm is described over the following three sections (Sections 3.2–3.4). Our algorithm is inspired by Gratzer's gluing-based argument for normalization [13] and we conjecture that this link can be made sufficiently precise to establish the soundness and completeness of our code.

▶ Remark 14. The version of normalization-by-evaluation we use is robust enough to require only local modifications in order to accommodate modal types. Accordingly, we focus primarily on connectives like dependent products and modal types whose behavior is impacted and refer the reader to, e.g., Abel [2] for a description how the algorithm works on the remaining connectives.

$$\begin{tikzcd}
 & \mathsf{Val} \arrow[d, "{\downarrow^{A}}"] & & \\
 & \mathsf{NfVal} \arrow[d, "\underline{\mathbf{quo}}(-)_n"] & \mathsf{Neutrals} \arrow[ul, "{\uparrow^{A}}"'] \arrow[d, "\underline{\mathbf{quo}}(-)_n"] & \mathsf{Level} \arrow[l, "\mathbf{q}_{n-k}"] \arrow[d, shift right] \\
\mathsf{Syntax} \arrow[uur, "{[\![-]\!]_{\rho}}"] & \mathsf{NfTerms} \arrow[l, hook, "{|-|}"] & \mathsf{NeTm} \arrow[l, hook, "\subseteq"] & \mathsf{Index} \arrow[l, "\mathbf{q}_k"] \arrow[u, shift right]
\end{tikzcd}$$

Figure 1 Overview of the algorithm inspired by [17] and [2].

3.1 The Domain

We start by a brief overview of the semantic domains described in Figure 1:

$$\begin{array}{llcl}
\textit{(values)} & A, u & ::= & {\uparrow^{A}} e \mid \lambda(f) \mid (\mu \mid A) \to B \mid \mathsf{zero} \mid \mathsf{suc}(v) \mid \mathsf{Nat} \mid (v_1, v_2) \\
 & & & \mid A \times B \mid \langle \mu \mid A \rangle \mid \mathsf{mod}_{\mu}(v) \\
\textit{(neutrals)} & e & ::= & \mathbf{q}_k \mid \mathsf{app}[\mu](e, d) \mid \mathsf{pr}_1(e) \mid \mathsf{pr}_2(e) \mid \mathsf{letmod}(\mu, \nu, C, c, A, e) \\
 & & & \mid \mathsf{rec}(C, u, v, e) \\
\textit{(environments)} & \rho & ::= & \cdot \mid \rho.v \\
\textit{(closures)} & C, f & ::= & \mathsf{clo}(M, \rho) \\
\textit{(normals)} & d & ::= & {\downarrow^{A}} v
\end{array}$$

Informally, *neutral forms* are generated by variables and elimination forms stuck on other neutrals. To a first approximation, a neutral is a chain of eliminations which are stuck on a variable. On the other hand, *values* – the codomain of the evaluation function – are primarily generated by introduction forms. In particular, there are no elimination forms directly available on values and there is no uniform way to turn a value into a neutral form. Consequently, β-reducible terms cannot be expressed in this grammar. One can, however, lift a neutral into a value after annotating the neutral form with its type. Tersely, values are β-short but not necessarily η-long.

A defining aspect of our approach to NbE is the handling of open terms. Rather than directly evaluating under a binder, when we reach, e.g., a lambda, we suspend the computation and store the intermediate result in a *closure*. The evaluation is resumed as soon as further information is gathered. In the case of a function, for instance, the evaluation of the body is resumed only after the function is applied. A closure is a combination of the term being evaluated and "the state of the evaluation algorithm." The latter amounts to the environment of variables which is reified and stored in the closure alongside the term.

Normal forms have only one constructor, *reification*. Values are lifted to normals by annotating them with a type. This type annotation is used during the quotation process in Section 3.3 in order to deal with the η-laws.

We emphasize that while terms use De Bruijn indices, neutral forms use De Bruijn *levels* to represent variables. This small maneuver ensures that values, neutral forms, and normal forms are silently weakened and we will capitalize on this fact throughout our algorithm [2].

3.2 Evaluation

Evaluation is the procedure of interpreting syntax into the semantic domains, specifically values. At a high-level, this amounts to β-reducing all terms (recall β-reducible terms cannot be represented as values). The presence of variables, however, causes some elimination forms to become stuck. These stuck terms are evaluated into neutrals and annotated with a type.

We single out a few interesting cases of the evaluation algorithm shown in Figure 2.

$$\boxed{[\![\_]\!]_{\_} : \mathsf{Syntax} \to \mathsf{Env} \to \mathsf{Val}}$$

$$\textsc{eval/var}\ \frac{\rho(i) = v}{[\![\mathbf{q}_i]\!]_\rho = v} \qquad \textsc{eval/pi}\ \frac{[\![A]\!]_\rho = A_0}{[\![(\mu \mid A) \to B]\!]_\rho = (\mu \mid A_0) \to \mathsf{clo}(B, \rho)} \qquad \textsc{eval/modify}\ \frac{[\![A]\!]_\rho = A_0}{[\![\langle \mu \mid A \rangle]\!]_\rho = \langle \mu \mid A_0 \rangle}$$

$$\textsc{eval/mod}\ \frac{[\![M]\!]_\rho = v}{[\![\mathsf{mod}_\mu(M)]\!]_\rho = \mathsf{mod}_\mu(v)} \qquad \textsc{eval/app}\ \frac{[\![M]\!]_\rho = u \qquad [\![N]\!]_\rho = v}{[\![M(N)_\mu]\!]_\rho = \underline{\mathbf{app}}(u, v)}$$

$$\textsc{eval/letmod}\ \frac{[\![M]\!]_\rho = v}{[\![\mathsf{let}_\nu\ \mathsf{mod}_\mu(\_) \leftarrow M\ \mathsf{in}\ N : A]\!]_\rho = \underline{\mathbf{letmod}}_{\nu;\mu}(\mathsf{clo}(A, \rho), \mathsf{clo}(N, \rho), v)}$$

$$(\rho.v)(0) = v \qquad\qquad (\rho.v)(i+1) = \rho(i)$$

Figure 2 Evaluation function, selected cases.

The work of evaluation is done around eliminators. Therefore, we single these cases out and define 'helper' functions for this portion of the algorithm. The interesting new cases are **letmod** and **app**, but generally for every syntax elimination form we define a suggestively named function that automatically beta-reduces eliminators applied to an introduction form, or returns a neutral and annotates it with its type.

$$\boxed{\underline{\mathbf{app}}(u, v) : \mathsf{Val} \qquad \underline{\mathbf{proj}}_i(v) : \mathsf{Val} \qquad \underline{\mathbf{letmod}}_{\nu;\mu}(C, c, v) : \mathsf{Val} \qquad \underline{\mathbf{J}}(C, c_{\mathsf{refl}}, p) : \mathsf{Val}}$$

$$\frac{}{\underline{\mathbf{inst}}(\mathsf{clo}(M, \rho), v) = [\![M]\!]_{\rho.v}} \qquad\qquad \frac{C = \lambda(C)}{\underline{\mathbf{app}}(u, v) = \underline{\mathbf{inst}}(C, v)}$$

$$\frac{u = {\uparrow^{A_0}} e \qquad A_0 = (\mu \mid A) \to C \qquad \underline{\mathbf{inst}}(C, v) = B}{\underline{\mathbf{app}}(u, v) = {\uparrow^{B}} \mathsf{app}[\mu](e, {\downarrow^{A}} v)} \qquad \frac{v = \mathsf{mod}_{\mu}(v') \qquad \underline{\mathbf{inst}}(c, v') = u}{\underline{\mathbf{letmod}}_{\nu;\mu}(C, c, v) = u}$$

$$\frac{v = {\uparrow^{A_0}} e \qquad A_0 = \langle \mu \mid A \rangle \qquad \underline{\mathbf{inst}}(C, {\uparrow^{\langle \mu \mid A \rangle}} e) = B}{\underline{\mathbf{letmod}}_{\nu;\mu}(C, c, v) = {\uparrow^{B}} \mathsf{letmod}(\nu, \mu, C, c, A, e)}$$

As mentioned previously, we use closures to represent syntax that cannot be evaluated in the present environment. Once we have found the value to complete the environment, we *instantiate* the closure with it and continue the evaluation in the extended environment.

3.3 Quotation

Quotation is the process of turning normals into terms. We will ensure that the results of quotation are always *normal form terms*, that is, β-short and η-long terms.

To account for the fact that normal forms mention values and neutral forms, quotation is split into three mutually recursive functions. Quotation must perform η-expansion and is therefore type-directed. Accordingly, while we have a quotation procedure which applies to values, this portion of the algorithm can only be used for quoting types where there is no associated η-expansions. All three of these functions take a natural number in addition to the actual term being quoted. This number represents the next available De Bruijn level for a free variable; it is used to quote terms with binders.

We present the novel cases of quotation of normal forms – those with modalities – below:

$$\frac{A_0 = (\mu \mid A) \to B \qquad \underline{\mathbf{inst}}(B, {\uparrow^{A}} \mathbf{q}_k) = B \qquad \underline{\mathbf{quo}}({\downarrow^{B}} \underline{\mathbf{app}}(v, {\uparrow^{A}} \mathbf{q}_k))_{k+1} = M}{\underline{\mathbf{quo}}({\downarrow^{A_0}} v)_k = \lambda(M)}$$

$$\frac{A_0 = \langle \mu \mid A \rangle \qquad v = \mathsf{mod}_{\mu}(w)}{\underline{\mathbf{quo}}({\downarrow^{A_0}} v)_k = \mathsf{mod}_{\mu}(\underline{\mathbf{quo}}({\downarrow^{A}} w)_k)} \qquad \frac{A_0 = \langle \mu \mid A \rangle \qquad v = {\uparrow^{B}} e}{\underline{\mathbf{quo}}({\downarrow^{A_0}} v)_k = \underline{\mathbf{quo}}(e)_k}$$

$$\frac{A_0 = {\uparrow^{A}} e \qquad v = {\uparrow^{A'}} e}{\underline{\mathbf{quo}}({\downarrow^{A_0}} v)_k = \underline{\mathbf{quo}}(e)_k}$$

We draw attention to one aspect of the first rule. This rule quotes a function, so consider the case where $v = \lambda(\mathsf{clo}(M, \rho))$. We create a fresh variable ${\uparrow^{A}} \mathbf{q}_k$ and make the semantic application $\underline{\mathbf{app}}(\lambda(\mathsf{clo}(M, \rho)), {\uparrow^{A}} \mathbf{q}_k)$. This last step is only sensible because values are closed under silent weakening; otherwise ρ would need to be weakened over $\mathbf{q}_k$.

Finally, we record the novel cases of quotation for neutral forms:

$$\frac{}{\underline{\mathbf{quo}}(\mathsf{app}[\mu](e, d))_k = \underline{\mathbf{quo}}(e)_k(\underline{\mathbf{quo}}(d)_k)_{\mu}}$$

$$\frac{\underline{\mathbf{inst}}(C, \mathsf{mod}_{\mu}({\uparrow^{A}} \mathbf{q}_k)) = B \qquad \underline{\mathbf{inst}}(c, {\uparrow^{A}} \mathbf{q}_k) = v}{\underline{\mathbf{quo}}(\mathsf{letmod}(\nu, \mu, C, c, A, e))_k = \mathsf{let}_{\nu}\ \mathsf{mod}_{\mu}(\_) \leftarrow \underline{\mathbf{quo}}(e)_k \ \mathsf{in}\ \underline{\mathbf{quo}}({\downarrow^{B}} v)_{k+1}}$$

3.4 The NbE function

Having defined both evaluation and quotation, we are almost in a position to define the complete normalization algorithm. The only missing step is the construction of the *initial environment* from a context. This portion of the algorithm takes a context Γ and produces

```
type mode                                  type m
val eq_mode : mode → mode → bool           val idm : m
                                           val compm : m → m → m
                                           val dom_mod : m → mode → mode
                                           val cod_mod : m → mode → mode
                                           val (=) : m → m → bool
                                           val (≤) : m → m → bool
```

Figure 3 A fragment of the signature for mode theories used in `mitten`.

an environment consisting of the variables bound in Γ. We then use this environment to kick off the evaluation of terms in context Γ:

$$\underline{\mathbf{reflect}}(\mathbf{1}) = \cdot \qquad\qquad \underline{\mathbf{reflect}}(\Gamma.(\mu \mid A)) = \underline{\mathbf{reflect}}(\Gamma).\uparrow^{[\![A]\!]_{\underline{\mathbf{reflect}}(\Gamma)}} \mathbf{q}_{|\Gamma|}$$

Finally, the complete normalization algorithm evaluates a term $\Gamma \vdash M : A \,@\, m$ in the initial environment specified by Γ and quotes it back:

$$\underline{\mathbf{norm}}_{\Gamma,A}(M) = \underline{\mathbf{quo}}(\downarrow^{[\![A]\!]_{\underline{\mathbf{reflect}}(\Gamma)}} [\![M]\!]_{\underline{\mathbf{reflect}}(\Gamma)})_{|\Gamma|}$$

4 Implementing a Mode Theory

Thus far we have been somewhat vague about which mode theory we were instantiating MTT with. The normalization algorithm given in Section 3, for instance, did not need to manipulate or compare modalities and so this point was easy to gloss over. The type-checker, on the other hand, must manipulate and scrutinize modalities and its definition requires a precise specification of a mode theory. Accordingly, we now present a representation of mode theories and operations upon them suitable for implementing a type-checker.

Concretely, our presentation follows the actual representation of mode theories used in `mitten`, our implementation of MTT. In `mitten`, all information specific to a mode theory is confined to a single OCaml module on which the type-checker depends. In particular, to configure `mitten` with a new mode theory, it is only necessary to implement that single module. There are three parts to our signature for mode theories (summarized in Figure 3):

1. Two abstract types; one for modes and one for modalities.
2. Various operations to compose modalities, extract the domain or codomain mode from a modality, or construct the identity modality.
3. Three operations to compare modes for equality and modalities for (in)equality.

It is these last two operations which are particularly crucial. Recall that not all mode theories admit decidable (in)equality and without it, type-checking MTT is undecidable. Accordingly, any implementation of MTT will require the user to supply a decision procedure for the mode theory. Our implementation shows that this information is both necessary and essentially sufficient. We note that the decision procedures for mode theories are completely separate from the terms and types of MTT and no knowledge of e.g., normalization-by-evaluation is required for their implementation.[2]

[2] See the following for examples: `https://github.com/logsem/mitten_preorder/blob/main/src/lib/`

▶ Remark 15. The reader might wonder why `idm` is not parametrized over `mode`. This is because `idm` internally is a placeholder for some identity modality, whose mode is elaborated. This alleviates practitioners of some tedious bookkeeping obligations in their proofs. This approach necessitates that the boundary projections `dom_mod` and `cod_mod` take an additional argument of type `mode`, which is returned on input `idm`. Essentially, we assume that always one part of the boundary is known so `dom_mod` gets a modality and its codomain as argument whereas `cod_mod` gets a modality and its domain as argument.

5 Semantic Type-Checking Algorithm

Having defined the normalization algorithm, we now define the type-checking algorithm for MTT. As mentioned in Sections 1 and 2, the algorithm is a variant of Coquand's semantic type checking algorithm for dependent type theory [11]. Accordingly, the algorithm breaks into two distinct phases: checking and synthesis. The checking portion of the algorithm accepts a context Γ, a term M, and a type A and checks that M has type A in context Γ. The synthesis phase accepts only the context and term, and synthesizes the type of the term in this context.

This simple picture is slightly complicated in the case of MTT, where various side conditions must be managed. For instance, we must ensure that the modalities a user writes in modal types are well-formed and that the term and type exist at the same mode as the context. These same considerations also require us to form a more intricate notion of a *semantic context* specifically for the type-checking algorithm.

We discuss the definition of semantic contexts in Section 5.1 and present a representative fragment of the type-checking algorithm itself in Section 5.2.

5.1 Semantic Contexts

In Section 2.2, we explained the intuitions behind MTT while working informally with the collection of variables in scope. Prior to discussing the type checker, we must describe the precise notion of context to organize these variables. Two factors complicate this otherwise standard structures: the modal annotations and restrictions and the need to evaluate terms during type-checking.

To a first approximation, contexts are still lists of variables with types but with additional bells and whistles added in order to support these two requirements. In order to record the necessary modal information, each variable is annotated by a modality. Deviating from Section 2.2, we add a new context operation $\Xi.\{\mu\}$ to "lazily" restrict all entries in a context Ξ by μ rather than storing this information on each variable separately.

For the second requirement, recall that type-checking must repeatedly test when two types are equal for the *conversion rule*. Accordingly, the context must store enough information to support this conversion test. We follow Coquand [11] and represent each type in the context by the corresponding *value* (in the sense of Section 3) and pair each variable with a corresponding value. This value may just be $\uparrow^{A} \mathbf{q}_i$, but it may also store the term associated definition. By storing information in this form, we can easily project out a *semantic environment* of a context and use that to evaluate a term and check for convertibility during type checking.

The grammar of *semantic contexts* is presented below:

$$\textit{(semantic contexts)} \quad \Xi \quad ::= \quad \cdot \mid \Xi \,.\, (v :_{\mu} A \mathbin{@} m) \mid \Xi.\{\mu\}$$

We now define two functions: The partial *lookup function*, which displays the type with its annotation and restriction as well as the mode it lives at, and the *stripping* function, which returns an environment by projecting out only the value components of the semantic context. The lookup function is undefined whenever a De Bruijn index is larger than the length of the context.

$$\begin{aligned}
&(\Xi\,.\,(v :_\mu A \mathbin{@} m))(0) = (\mu|A)_m, \{\mathsf{id}\} \\
&(\Xi\,.\,(w :_\xi B \mathbin{@} o))(i+1) = (\mu|A)_m, \{\nu\} && \text{where } (\mu|A)_m, \nu = \Xi(i) \\
&(\Xi.\{\nu'\})(i) = (\mu|A)_m, \{\nu \circ \nu'\} && \text{where } (\mu|A)_m, \nu = \Xi(i) \\
&|\cdot| = \cdot \\
&|\Xi\,.\,(v :_\mu A \mathbin{@} m)| = |\Xi|.v \\
&|\Xi.\{\mu\}| = |\Xi|
\end{aligned}$$

▶ **Notation 16.** *If we extend a semantic context with a type where the value is a fresh variable, we hide it to make the expression more readable.*

$$\Xi.(\mu|A) \triangleq \Xi\,.\,(\uparrow^A \mathbf{q}_k :_\mu A \mathbin{@} m) \qquad \textit{where } k = \mathsf{length}(\Xi)$$

If the modality μ is furthermore the identity modality, we omit it and write

$$\Xi.A \triangleq \Xi\,.\,(\uparrow^A \mathbf{q}_k :_{\mathsf{id}} A \mathbin{@} m) \qquad \textit{where } k = \mathsf{length}(\Xi)$$

5.2 Checking and Synthesis

We now come to the type-checking algorithm which is split into a pair of judgments: $\Xi \vdash M \Leftarrow A \mathbin{@} m$ and $\Xi \vdash R \Rightarrow A \mathbin{@} m$. The first, $\Xi \vdash M \Leftarrow A \mathbin{@} m$, handles type checking which tests if M has type A in Ξ. The second, $\Xi \vdash M \Rightarrow A \mathbin{@} m$, implements type synthesis and accordingly takes only the semantic context Ξ and term M and returns type A of M in context Ξ if one can be inferred.

We present a few representative rules for these judgments (and explain them below). To ensure that terms and types are well-formed, we utilize the functions exposed by the signature presented in Section 4. In particular, $n \overset{?}{=} m$ checks whether two modes are equal and $\mu \leq \nu$ is the modality ordering relation. Furthermore, with $\mu.\mathsf{dom}$ and $\mu.\mathsf{cod}$ we denote the respective domain and codomain of a modality – denoted `dom_mod` and `cod_mod` respectively in Section 4. For readability, we leave the second argument of $\mu.\mathsf{dom}$ and $\mu.\mathsf{cod}$ implicit.

$$\textsc{Pi}\quad \frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathsf{U} \mathbin{@} \mu.\mathsf{dom} \qquad \Xi.(\mu|A) \vdash B \Leftarrow \mathsf{U} \mathbin{@} m \qquad \mu.\mathsf{cod} \overset{?}{=} m}{\Xi \vdash (\mu \mid A) \to B \Leftarrow \mathsf{U} \mathbin{@} m}$$

$$\textsc{Mod-Form}\quad \frac{\Xi.\{\mu\} \vdash A \Leftarrow \mathsf{U} \mathbin{@} \mu.\mathsf{dom} \qquad \mu.\mathsf{cod} \overset{?}{=} m}{\Xi \vdash \langle \mu \mid A \rangle \Leftarrow \mathsf{U} \mathbin{@} m}
\qquad
\textsc{Mod-Intro}\quad \frac{\Xi.\{\mu\} \vdash M \Leftarrow A \mathbin{@} \mu.\mathsf{dom} \qquad \mu.\mathsf{cod} \overset{?}{=} m}{\Xi \vdash \mathtt{mod}_\mu(M) \Leftarrow \langle \mu \mid A \rangle \mathbin{@} m}$$

$$\textsc{Conv}\quad \frac{\Xi \vdash R \Rightarrow B \mathbin{@} m \qquad A \equiv_{|\Xi|} B}{\Xi \vdash R \Leftarrow A \mathbin{@} m}
\qquad
\textsc{Var}\quad \frac{\Xi(k) = (\mu|A)_m, \nu \qquad \mu \leq \nu \qquad m \overset{?}{=} n}{\Xi \vdash \mathbf{q}_k \Rightarrow A \mathbin{@} n}$$

$$\frac{\begin{array}{c}\nu.\mathsf{cod} \overset{?}{=} m \qquad \Xi.\{\nu\} \vdash R \Rightarrow \langle \mu \mid A \rangle \mathbin{@} \nu.\mathsf{dom} \qquad \Xi.(\nu, \langle \mu \mid A \rangle) \vdash C \Leftarrow \mathsf{U} \mathbin{@} m \\ k = \mathsf{length}(\Xi) \qquad \Xi.(\nu \circ \mu, A) \vdash N \Leftarrow [\![C]\!]_{|\Xi|.\mathsf{mod}_\mu(\uparrow^{A}\mathbf{q}_k)} \mathbin{@} m\end{array}}{\Xi \vdash \mathtt{let}_\nu\ \mathtt{mod}_\mu(\_) \leftarrow R\ \mathtt{in}\ N\ \mathtt{over}\ C \Rightarrow [\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}} \mathbin{@} m}\ \textsc{mod-elim}$$

We first consider the formation rule for dependent products. First we verify that indeed $\mu.\mathsf{cod} \overset{?}{=} m$ to ensure that the modality μ can be used at this mode. Recall that Π-types in MTT go from a μ-restricted type A to a non restricted type B. Accordingly, we check that A is a type in the μ-restricted semantic context $\Xi.\{\mu\}$ and that B is well-formed in the context $\Xi.(\mu|A)$. Note that when checking A we change the mode to $\mu.\mathsf{dom}$.

Since the modal formation and introduction rules follow a similar pattern we will only look at the modal introduction rule. To validate that $\mathtt{mod}_\mu(M)$ has type $\langle \mu \mid A \rangle$ at mode m we first verify that $\mu.\mathsf{cod} \overset{?}{=} m$. Then we check that M has type A in the μ-restricted environment $\Xi.\{\mu\}$ at mode $\mu.\mathsf{dom}$.

Next, we discuss the conversion rule. When considering a synthesizable term R, the type-checking algorithm proceeds somewhat differently. We first synthesize the type of R and then compare the result to the type we were given to check R against. It is this comparison which uses the normalization algorithm of Section 3 to compute the normal forms of A and B and decides afterwards the equality of the normalized expressions.

To synthesize a variable $\mathbf{q}_k$ in a semantic context Ξ at mode m we first compute the type of the variable together with its annotation and restriction $(\mu|A)_m, \{\nu\}$, using the lookup function defined in Section 5.1. Before we return A as the type of $\mathbf{q}_k$, we must also perform an additional check to ensure that $\mu \leq \nu$ so that this occurrence of the variable is valid.

Finally, we consider the modal elimination case. Recall from Section 2.2 that the modal elimination principle allows us to 'pattern-match' on a term $R : \langle \mu \mid A \rangle$ in a ν-restricted context and replace it with a variable $x :_{\nu \circ \mu} A$. To synthesize $\mathtt{let}_\nu\ \mathtt{mod}_\mu(\_) \leftarrow R\ \mathtt{in}\ N\ \mathtt{over}\ C$, we take advantage of the fact that the user provides the motive C already; if this term is well-typed, its type must be $[\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}}$.

There are, however, several checks to perform to ensure that the term is actually well-typed. First, we check that $\nu.\mathsf{cod} \overset{?}{=} m$. Next, we synthesize the type of R in the ν-restricted context and check that the result is of the form $\langle \mu \mid A \rangle$. Having computed $\langle \mu \mid A \rangle$, we then check that both C and N are well-formed. The motive C must be a type in the extended context $\Xi.(\nu, \langle \mu \mid A \rangle)$ while N must have type $[\![C]\!]_{|\Xi|.[\![R]\!]_{|\Xi|}}$ in context $\Xi.(\nu, \langle \mu \mid A \rangle)$.

A complete implementation of the algorithm can be found at `https://github.com/logsem/mitten_preorder/blob/main/src/lib/check.ml`.

6 Case study: guarded recursion in `mitten`

We now discuss an extended example using `mitten` with a particular choice of mode theory. By instantiating `mitten` appropriately, we convert it into a proof assistant for *guarded recursion* and use it to reason about classical examples from the theory.

6.1 Guarded recursion

Guarded recursion provides a discipline for managing recursive definitions within type theory without compromising soundness. In particular, guarded type theory extends type theory with a handful of modalities (►, Γ and Δ) along with a modified version of the fixed-point combinator:

$$\mathsf{loeb} : (\blacktriangleright A \to A) \to A$$

By placing the recursive call under a $\blacktriangleright$, this weakened fixed-point combinator does not result in inconsistencies. Together with the other modalities, moreover, it can be used to define and reason about coinductive types and gives rise to a *synthetic* form of domain theory.

Following [8], we are interested in using `mitten` as a tool to reason about a particular *model* of guarded recursion: $\mathbf{PSh}(\omega)$. In fact, using MTT's capacity to reason about multiple categories at once, we will work with a slightly richer model which includes both $\mathbf{PSh}(\omega)$ and $\mathbf{Set}$. In this model, the aforementioned modalities are all interpreted by right adjoints:

$$\begin{array}{lll} \Gamma : \mathbf{PSh}(\omega) \to \mathbf{Set} & \Delta : \mathbf{Set} \to \mathbf{PSh}(\omega) & \blacktriangleright : \mathbf{PSh}(\omega) \to \mathbf{PSh}(\omega) \\ \Gamma(X) = [\mathbf{1}, X] & \Delta(S) = \lambda\_.\ S & \blacktriangleright(X)(0) = \{\star\} \quad \blacktriangleright(X)(n+1) = X(n) \end{array}$$

In particular, the composite of Γ and Δ is the global sections comonad $\square$. The fixed-point operator loeb in $\mathbf{PSh}(\omega)$ is definable using induction over ω.

Gratzer et. al [15] have shown that MTT with a mode theory axiomatizing these three modalities is modeled by these two categories and therefore provides a suitable basis for guarded recursion. We recall their mode theory in Figure 4.

$$\begin{tikzcd} t \arrow[loop left, "\ell"] \arrow[r, bend left, "\gamma"] & s \arrow[l, bend left, "\delta"] \end{tikzcd}$$

$$\begin{array}{rr} \delta \circ \gamma \leq 1 & 1 = \gamma \circ \delta \\ 1 \leq \ell & \gamma = \gamma \circ \ell \\ \mu \leq \nu \wedge \nu \leq \mu & \implies \ \mu = \nu \end{array}$$

Figure 4 $\mathcal{G}$: a mode theory for guarded recursion.

The equalities represented in Figure 4 together with the equational theory of MTT ensure that $\square = \delta \circ \gamma$ is an idempotent comonad and that the following equivalence is definable:

$$\langle \square \mid \langle \ell \mid A \rangle \rangle \simeq \langle \square \mid A \rangle.$$

In order to actually reason about guarded definitions, however, we still must add Löb induction to the system. Adding Löb induction primitively raises substantial issues [14], so we opt to axiomitize it along with a (propositional) equation specifying its unfolding principle:

$$\mathsf{loeb} : ((\ell \mid A) \to A) \to A \mathbin{@} t \qquad \mathsf{unfold} : (f : (\ell \mid A) \to A) \to \mathsf{Id}_A(\mathsf{loeb}\, f, f(\mathsf{loeb}\, f)) \mathbin{@} t$$

As to be expected, these new constants disrupt canonicity but crucially cause no issues for type checking. We now discuss how to instantiate `mitten` with this particular mode theory.

6.2 Implementation

In order to use `mitten` to reason about guarded MTT, we must construct an implementation of the mode theory module corresponding to Figure 4 and extend `mitten` with constants for Löb induction. The latter point is routine; `mitten` supports adding axioms to a development. We therefore focus on the first step: the implementation of the mode theory.

The main challenge when implementing Figure 4 is to show that the relation $\leq$ is decidable. We have done so by using a (simple) form of normalization-by-evaluation to reduce modalities in this mode theory to normal forms which can be directly compared.

▶ Remark 17. We leave the modes during the evaluation implicit and assume, without loss of generality, that we are only considering well-formed modalities.[3]

By studying the category generated by Figure 4, it becomes clear that $\mathcal{G}$ is far from a free mode theory. In fact, many possible compositions trivialize; in a chain of composable modalities we can freely remove any $\gamma \circ \delta$ as well as any ℓ to the right of a γ. Accordingly, there are only four kinds of expressions remaining which thus constitute *normal modalities*:

$$\textit{(Normal modalities)} \quad \mu, \nu \quad ::= \quad \ell^k \mid \ell^k \circ \delta \mid \ell^k \circ \delta \circ \gamma \mid \gamma \mid \mathsf{id}_s$$

Note that $k = 0$ is allowed and thus in particular $\delta \circ \gamma = \ell^0 \circ \delta \circ \gamma$ as well as $\mathsf{id} = \ell^0$. There is an evident map i sending a normal form μ to a modality in $\mathcal{G}$. We now construct an inverse to this map:

$$\begin{aligned}
\underline{\mathbf{eval}}(\mathsf{id}_t) &= \ell^0\\
\underline{\mathbf{eval}}(\mathsf{id}_s) &= \mathsf{id}_s\\
\underline{\mathbf{eval}}(\ell \circ \nu) &= \underline{\mathbf{comp}}(\ell, \underline{\mathbf{eval}}(\nu))\\
\underline{\mathbf{eval}}(\gamma \circ \nu) &= \underline{\mathbf{comp}}(\gamma, \underline{\mathbf{eval}}(\nu))\\
\underline{\mathbf{eval}}(\delta \circ \nu) &= \underline{\mathbf{comp}}(\delta, \underline{\mathbf{eval}}(\nu))
\end{aligned}$$

$$\begin{aligned}
\underline{\mathbf{comp}}(\ell, \ell^k) &= \ell^{k+1}\\
\underline{\mathbf{comp}}(\ell, \ell^k \circ \delta) &= \ell^{k+1} \circ \delta\\
\underline{\mathbf{comp}}(\ell, \ell^k \circ \delta \circ \gamma) &= \ell^{k+1} \circ \delta \circ \gamma\\
\underline{\mathbf{comp}}(\gamma, \ell^k) &= \gamma\\
\underline{\mathbf{comp}}(\gamma, \ell^k \circ \delta \circ \gamma) &= \gamma\\
\underline{\mathbf{comp}}(\gamma, \ell^k \circ \delta) &= \mathsf{id}_s\\
\underline{\mathbf{comp}}(\delta, \mathsf{id}_s) &= \ell^0 \circ \delta\\
\underline{\mathbf{comp}}(\delta, \gamma) &= \ell^0 \circ \delta \circ \gamma
\end{aligned}$$

▶ **Theorem 18.** *For any modality μ we have that $\mu = i(\underline{\mathbf{eval}}(\mu))$.*

Next, we define a (decidable)partial ordering on normal modalities:

$$\frac{m \le n}{\ell^m \sqsubseteq \ell^n} \qquad \frac{}{\gamma \sqsubseteq \gamma} \qquad \frac{m \le n}{\ell^m \circ \delta \circ \gamma \sqsubseteq \ell^n \circ \delta \circ \gamma} \qquad \frac{m \le n}{\ell^m \circ \delta \circ \gamma \sqsubseteq \ell^n} \qquad \frac{m \le n}{\ell^m \circ \delta \sqsubseteq \ell^n \circ \delta}$$

$$\frac{}{\mathsf{id}_s \sqsubseteq \mathsf{id}_s}$$

▶ **Theorem 19.** *For any normal modalities μ and ν we have $\mu \sqsubseteq \nu$ if and only if $i(\mu) \le i(\nu)$.*

▶ **Corollary 20.** *Equality of modes and inequality of modalities are both decidable.*

6.3 Streams in guarded `mitten`

We now illustrate the use of this instantiation of `mitten` by defining the types of guarded and coinductive streams and constructing various examples.

▶ Remark 21. In the following we deviate from our surface syntax to enhance readibility of the derivations. Thus, we leave many arguments implicit and alter certain notations. In particular, propositional identites are denoted by $a \equiv b$ instead of $\mathsf{Id}_A(a, b)$ and implicit arguments are omitted. We furthermore hide the type family C of the modal elimination rule in the following constructions.

We begin with the type of *guarded streams.*

$$\begin{aligned}
&\mathsf{gstream\_fun} : \mathsf{U} \to (\ell \mid \mathsf{U}) \to \mathsf{U} \mathbin{@} t\\
&\mathsf{gstream\_fun}\, A\, X = A \times \langle \ell \mid X \rangle
\end{aligned}
\qquad\qquad
\begin{aligned}
&\mathsf{gstream} : \mathsf{U} \to \mathsf{U} \mathbin{@} t\\
&\mathsf{gstream}\, A = \mathsf{loeb}(\mathsf{gstream\_fun}\, A)
\end{aligned}$$

[3] This assumption is justified since `mitten` checks all modalities prior to normalization and type-checking.

▶ **Notation 22.** *We will make use of several standard functions for intensional identity types such as the functions* $\mathsf{transport} : A \equiv B \to A \to B$ *and* $-^{-1} : a \equiv b \to b \equiv a$*.*

Recall that we have added Löb induction only with a *propositional* unfolding rule. Accordingly, we must use transport along this equality to obtain the folding and unfolding operations for $\mathsf{gstream}$:

$$\mathsf{gfold} : (A : \mathsf{U}) \to A \times \langle \ell \mid \mathsf{gstream}\, A \rangle \to \mathsf{gstream}\, A \mathbin{@} t$$
$$\mathsf{gfold}\, A = \mathsf{transport}\, (\mathsf{unfold}(\mathsf{gstream\_fun}\, A))^{-1}$$

$$\mathsf{gunfold} : (A : \mathsf{U}) \to \mathsf{gstream}\ A \to A \times \langle \ell \mid \mathsf{gstream}\, A \rangle \mathbin{@} t$$
$$\mathsf{gunfold}\, A = \mathsf{transport}\, (\mathsf{unfold}\, (\mathsf{gstream\_fun}\ A))$$

We are able to deduce the following equalities by using the fact that $\mathsf{transport}\, p$ is inverse to $\mathsf{transport}\, p^{-1}$:

$$\mathsf{fold\_unfold} : (s : \mathsf{gstream}\, A) \to \mathsf{gfold}\, A\, (\mathsf{gunfold}\, A\, s) \equiv s \mathbin{@} t$$
$$\mathsf{unfold\_fold} : (s : A \times \langle \ell \mid \mathsf{gstream}\, A \rangle) \to \mathsf{gunfold}\, A\, (\mathsf{gfold}\, A\, s) \equiv s \mathbin{@} t$$

Using this we can define the familiar operations on guarded streams and prove their expected equations.

$$\mathsf{ghead} : \mathsf{gstream}\, A \to A \qquad \mathsf{gtail} : \mathsf{gstream}\, A \to \langle \ell \mid \mathsf{gstream}\ A \rangle$$
$$\_ : \mathsf{gtail}(\mathsf{gcons}\, a\, s) \equiv s \qquad \mathsf{gcons} : A \to \langle \ell \mid \mathsf{gstream}\, A \rangle \to \mathsf{gstream}\, A$$
$$\_ : \mathsf{ghead}(\mathsf{gcons}\, a\, s) \equiv a \qquad \_ : \mathsf{gcons}\, (\mathsf{ghead} s)\, (\mathsf{gtail} s) \equiv s$$

With Löb induction, these definitions and equalities allow us to construct and work with *guarded* streams, which differ from coinductive streams in several important ways. For instance, the tail operation on guarded streams produces a guarded stream under a later which prevents us from writing an operation dropping every element of a guarded stream.

By making use of the other modalities of Figure 4, we are able to define the type of *coinductive* streams. To do so, we will use the following operations:

$$\mathsf{comp}_{\gamma,\delta} : \langle \gamma \mid \langle \delta \mid A \rangle \rangle \to A \qquad \mathsf{comp}_{\gamma,\ell} : \langle \gamma \mid \langle \ell \mid A \rangle \rangle \to \langle \gamma \mid A \rangle$$

Both of these are instances of the general composition principle for modalities available in MTT. We now define streams as follows:

$$\mathsf{stream} : \mathsf{U} \to \mathsf{U} \mathbin{@} s$$
$$\mathsf{stream}\, A = \langle \gamma \mid \mathsf{gstream}\, \langle \delta \mid A \rangle \rangle$$

$$\mathsf{head} : \mathsf{stream}\, A \to A \qquad \mathsf{tail} : \mathsf{stream}\, A \to \mathsf{stream}\, A$$
$$\mathsf{head}\, s = \qquad \mathsf{tail}\, s =$$
$$\quad \mathtt{let}_{\mathsf{id}}\ \mathtt{mod}_\gamma(g) = s\ \mathtt{in} \qquad \quad \mathtt{let}_{\mathsf{id}}\ \mathtt{mod}_\gamma(g) = s\ \mathtt{in}$$
$$\quad \mathsf{comp}_{\gamma,\delta}(\mathtt{mod}_\gamma(\mathsf{ghead}\, g)) \qquad \quad \mathsf{comp}_{\gamma,\ell}(\mathtt{mod}_\gamma(\mathsf{gtail}\, g))$$

We emphasize that the type of coinductive streams lives at mode s, the mode modeled by sets. Intuitively, by taking the global sections of a guarded stream we obtain the normal coinductive stream [10]. Indeed, using guarded recursion in mode t, we are able to equip this type with a coiteration principle:

$$\mathsf{go} : (\delta \mid A : \mathsf{U})(\delta \mid S : \mathsf{U})(\delta \mid S \to A \times S) \to (\delta \mid S) \to \mathsf{gstream}\, \langle \delta \mid A \rangle \mathbin{@} t$$
$$\mathsf{go}\, A\, S\, f = \mathsf{loeb}\, \lambda g\, s.\, \mathsf{gcons}(\mathsf{mod}_\delta(\pi_1\, (f\, s)), \mathsf{mod}_\ell(g\, (\pi_2\, (f\, s))))$$

$\mathsf{coiter} : (A : \mathsf{U})(S : \mathsf{U}) \to (S \to A \times S) \to S \to \mathsf{stream}\, A \mathbin{@} s$
$\mathsf{coiter}\, A\, S\, f\, s = \mathtt{mod}_{\gamma}(\mathsf{go}\, A\, S\, f\, s)$

Informally, this coiteration scheme induces a map from any $(A \times -)$-coalgebra to $\mathsf{stream}\, A$.

It is natural to wonder whether $\mathsf{stream}\, A$ is the *final coalgebra* for $(A \times -)$. In the presence of equality reflection, this was established by Gratzer et al. [15]. To replay this proof in `mitten`, we would require two ingredients not presently available: function extensionality and modal extensionality. The first is unsurprising, so we focus on the second. Modalities do not necessarily preserve identity types and therefore in general we cannot have a function:

$$(\ell \mid \mathsf{Id}_A(a, b)) \to \mathsf{Id}_{\langle \ell | A \rangle}(\mathtt{mod}_{\ell}(a), \mathtt{mod}_{\ell}(b))$$

Such a map is crucial to establish arguments of equality by Löb induction like the finality of $\mathsf{stream}\, A$. Having said this, we emphasize that without disrupting normalization we can extend MTT with a crisp induction principle which enables us to construct such a map and prove it to be an equivalence [13]. In the presence of this additional structure – or a postulate to the same effect – we conjecture that $\mathsf{stream}\, A$ is the final coalgebra.

We conclude with a simple example of the coiteration: the stream of all natural numbers.

$\mathsf{nats} : \mathsf{stream}\, \mathtt{Nat}$
$\mathsf{nats} = \mathsf{coiter}\, (\lambda n.\ (n, \mathtt{succ}(n)))\, 0$

7 Related Work

Modal proof assistants have seen a great deal of attention in the last several years. We compare our work on `mitten` to several of the most closely related lines of research.

Normalization for MTT

In [13], Gratzer proves that MTT enjoys a normalization algorithm. While his proof avoids a number of technicalities by adopting a synthetic approach to normalization, this obstructs extracting an actual algorithm for use in implementation. We have taken this next step and, inspired by the synthetic proof of normalization, obtained an actual algorithm suitable for implementation in the particular case of preordered mode theories. Furthermore, while Gratzer works relative to the assumption that the ambient mode theory is decidable, we have isolated the precise requirements necessary on the mode theory and shown that they are sufficiently flexible to accommodate common mode theories.

Alternative modal type theories

As already discussed, `mitten` implements a version of MTT [15] but many other modal type theories exist in the literature. For instance, de Paiva and Ritter, Shulman, Zwanziger and others [12, 24, 28] have studied dependent versions of *dual-context modal type theories*. We have chosen to focus on MTT over a dual-context system in order to capitalize on the normalization theorem proven for MTT as well as the greater degree of generality offered by the system. In particular, while dual-context type theories offer a convenient syntax for one modality or a pair of adjoint modalities, they do not easily adapt to incorporating multiple distinct modalities as we required in Section 6. Independently, Bahr et al., Birkedal et al, and others [4, 7, 17, 18] have experimented with modal type theories based upon *dependent right adjoints*. While these offer a potentially convenient syntax and normalization results for some theories have been established [17, 18], they are equally difficult to adapt to multiple modalities. Thus, while a wide variety of modal dependent type theories have been proposed in the literature, we feel that MTT offers a unique position in the space because of its generality and metatheory.

Proof assistants for a single modality

There have been multiple attempts to extend proof assistants with a single specific modality. Notably Vezzosi [27] extends Agda with an idempotent comonad and Gratzer et al. [17] created a proof assistant based around a similar modality. Agda builds on the aforementioned dual-context type theories while Gratzer et al. use a system based on dependent right adjoints. Both of these proof assistants are closely related to mitten. Concretely, the former may be encoded within mitten and mitten builds on the same core algorithms [11, 1] as the latter. Importantly, however, unlike these implementations mitten is not tied to a particular modal situation and can be easily adapted to accommodate other modalities.

Guarded recursion in Agda

In Section 6 we discussed an instantiation of mitten for guarded recursion. For this specific case, an experimental Agda extension is available [25]. This extension implements a version of clocked cubical type theory [6]. This variant of guarded type theory offers finer-grained guarded programming by exposing multiple independent later modalities; these can be used to interleave guarded types without issue. Furthermore, clocked cubical type theory capitalizes on certain primitives of cubical type theory to expose some definitional equalities around Löb induction. Guarded cubical Agda builds upon Agda's existing facilities for interactive proof developments and the system has been used for non-trivial developments [22, 26].

As a consequence of this more intricate theory, however, the metatheory of guarded cubical Agda is far less developed than the theory of mitten. Moreover, the infrastructure of guarded cubical Agda is (necessarily) specialized to just one modal situation. While mitten is a more primitive system than guarded cubical Agda, it is therefore far more flexible and offers a theoretical framework for many modal systems rather than being specialized to one.

Sikkel

Recently, Ceulemans et al. [9] have explored an alternative strategy for implementing MTT in Sikkel. Rather than constructing a custom proof assistant like mitten, they have provided a DSL for a simply-typed version of MTT within Agda. Within this DSL, one may construct terms in MTT which then compile to elements of an appropriate denotational semantics expressed within Agda. A major advantage of such an approach is the low startup cost: the full resources of the Agda proof assistant are available when working within Sikkel. By embedding within Agda, however, Sikkel's interface is less convenient and it is currently restricted only to simple types. Accordingly, we believe that a proof assistant designed for MTT from its inception offers a more promising route for serious modal programming.

Menkar

Menkar [23] is an earlier attempt at a proof assistant for multimodal programming developed by Nuyts. It predates – and in fact partially inspires – MTT, but contains both theoretical and practical deficiencies which led to its development being suspended in 2019. Inspired by the advances in proof theory for multimodal type theory obtained since Menkar's development, both mitten and Sikkel are early attempts to develop a theoretically sound replacement for Menkar. While not as fully-featured as Menkar, mitten in particular is an attempt to develop a principled modal proof assistant.

8 Conclusions and future work

We contribute mitten, a flexible proof assistant which can be specialized to a wide range of modal type theories. We have designed normalization and type-checking algorithms for mitten based on recent advances in the metatheory of MTT [13]. Finally, we have argued for mitten's utility by instantiating it to a mode theory suitable for guarded recursion and constructing various classical examples of guarded programs.

Thus far, mitten is restricted to working with preordered mode theories. While this constitutes a large and important class of examples, it would be desirable to implement full MTT and allow for arbitrary 2-categories as mode theories. Such an extension, however, would require a more refined normalization algorithm.

In particular, in our algorithm we have taken advantage of the absence of distinct 2-cells to avoid annotating variables with modal coercions. This, in turn, preserves a crucial invariant of NbE: it is never necessary to explicitly substitute within a value. Indeed, in our style of NbE such substitutions are not even possible; our representation of closures essentially precludes them. We hope to generalize our approach to cover full MTT by incorporating some techniques recently used by Hu and Pientka [18] in a normalization algorithm for a particular modal type theory. Essentially, they enable a small amount of substitution to occur during the normalization algorithm; by carefully structuring the necessary modal substitutions they are able to adapt the standard normalization-by-evaluation to their setting. We hope to do the same in mitten by generalizing their approach to support multiple interacting modalities.

References

1 Andreas Abel. Polarised subtyping for sized types. *Mathematical Structures in Computer Science*, 18(5):797–822, 2008. doi:10.1017/S0960129508006853.

2 Andreas Abel. *Normalization by Evaluation: Dependent Types and Impredicativity.* Habilitation, Ludwig-Maximilians-Universität München, 2013.

3 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi:10.1145/3158111.

4 Patrick Bahr, Hans Bugge Grathwohl, and Rasmus Ejlers Møgelberg. The clocks are ticking: No more delays! In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*. IEEE, 2017. doi:10.1109/LICS.2017.8005097.

5 Patrick Bahr, Christian Uldal Graulund, and Rasmus Ejlers Møgelberg. Simply ratt: A fitch-style modal calculus for reactive programming without space leaks. *Proc. ACM Program. Lang.*, 3:109:1–109:27, 2019. doi:10.1145/3341713.

6 Magnus Baunsgaard Kristensen, Rasmus Ejlers Mogelberg, and Andrea Vezzosi. Greatest hits: Higher inductive types in coinductive definitions via induction under clocks. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3531130.3533359.

7 Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal dependent type theory and dependent right adjoints. *Mathematical Structures in Computer Science*, 30(2):118–138, 2020. doi:10.1017/S0960129519000197.

8 Lars Birkedal, Rasmus Møgelberg, Jan Schwinghammer, and Kristian Støvring. First steps in synthetic guarded domain theory: step-indexing in the topos of trees. *Logical Methods in Computer Science*, 8(4), 2012. doi:10.2168/LMCS-8(4:1)2012.

9 Joris Ceulemans, Andreas Nuyts, and Dominique Devriese. Sikkel: Multimode simple type theory as an agda library. In *Electronic Proceedings in Theoretical Computer Science*, volume 360, pages 93–112. Munich, Germany, Open Publishing Association, June 2022. doi:10.4204/EPTCS.360.5.

10 Ranald Clouston, Aleš Bizjak, Hans Bugge Grathwohl, and Lars Birkedal. Programming and reasoning with guarded recursion for coinductive types. In Andrew Pitts, editor, *Foundations of Software Science and Computation Structures*, pages 407–421. Springer Berlin Heidelberg, 2015.

11 Thierry Coquand. An algorithm for type-checking dependent types. *Science of Computer Programming*, 26(1):167–177, 1996. doi:10.1016/0167-6423(95)00021-6.

12 Valeria de Paiva and Eike Ritter. Fibrational modal type theory. In *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, 2015. doi:10.1016/j.entcs.2016.06.010.

13 Daniel Gratzer. Normalization for multimodal type theory. In *Proceedings of the 37th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '22, New York, NY, USA, 2022. Association for Computing Machinery. doi:10.1145/3531130.3532398.

14 Daniel Gratzer and Lars Birkedal. A Stratified Approach to Löb Induction. In Amy P. Felty, editor, *7th International Conference on Formal Structures for Computation and Deduction (FSCD 2022)*, volume 228 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:22, Dagstuhl, Germany, 2022. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.FSCD.2022.23.

15 Daniel Gratzer, G. A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal Dependent Type Theory. *Logical Methods in Computer Science*, Volume 17, Issue 3, July 2021. doi:10.46298/lmcs-17(3:11)2021.

16 Daniel Gratzer, G.A. Kavvos, Andreas Nuyts, and Lars Birkedal. Multimodal dependent type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20. ACM, 2020. doi:10.1145/3373718.3394736.

17 Daniel Gratzer, Jonathan Sterling, and Lars Birkedal. Implementing a Modal Dependent Type Theory. *Proc. ACM Program. Lang.*, 3, 2019. doi:10.1145/3341711.

18 Jason Z. S. Hu and Brigitte Pientka. An investigation of kripke-style modal type theories, 2022. doi:10.48550/arXiv.2206.07823.

19 Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. In H. Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, Leibniz International Proceedings in Informatics (LIPIcs), pages 22:1–22:17. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2018. doi:10.4230/LIPIcs.FSCD.2018.22.

20 Daniel R. Licata and Michael Shulman. Adjoint Logic with a 2-Category of Modes. In Sergei Artemov and Anil Nerode, editors, *Logical Foundations of Computer Science*, pages 219–235. Springer International Publishing, 2016. doi:10.1007/978-3-319-27683-0_16.

21 Daniel R. Licata, Michael Shulman, and Mitchell Riley. A Fibrational Framework for Substructural and Modal Logics. In Dale Miller, editor, *2nd International Conference on Formal Structures for Computation and Deduction (FSCD 2017)*, volume 84 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 25:1–25:22. Schloss Dagstuhl – Leibniz-Zentrum fuer Informatik, 2017. doi:10.4230/LIPIcs.FSCD.2017.25.

22 Rasmus Ejlers Møgelberg and Niccolò Veltri. Bisimulation as path type for guarded recursive types. *Proc. ACM Program. Lang.*, 3(POPL), January 2019. doi:10.1145/3290317.

23 Andreas Nuyts. Menkar. https://github.com/anuyts/menkar, 2019.

24 Michael Shulman. Brouwer's fixed-point theorem in real-cohesive homotopy type theory. *Mathematical Structures in Computer Science*, 28(6):856–941, 2018. doi:10.1017/S0960129517000147.

25 The Agda Team. Agda, 2022. URL: https://agda.readthedocs.io/en/latest/language/guarded-cubical.html.

26 Niccolò Veltri and Andrea Vezzosi. Formalizing π-calculus in guarded cubical agda. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, pages 270–283, 2020.

27 Vezzosi, Andrea. `agda-flat`, 2018. URL: `https://github.com/agda/agda/tree/flat`.

28 Colin Zwanziger. Natural model semantics for comonadic and adjoint type theory: Extended abstract. In *Preproceedings of Applied Category Theory Conference 2019*, 2019.

An Irrelevancy-Eliminating Translation of Pure Type Systems

Nathan Mull
University of Chicago, IL, USA

Abstract

I present an infinite-reduction-path-preserving typability-preserving translation of pure type systems which eliminates rules and sorts that are in some sense irrelevant with respect to normalization. This translation can be bootstrapped with existing results for the Barendregt-Geuvers-Klop conjecture, extending the conjecture to a larger class of systems. Performing this bootstrapping with the results of Barthe et al. [4] yields a new class of systems with dependent rules and non-negatable sorts for which the conjecture holds. To my knowledge, this is the first improvement in the state of the conjecture since the results of Roux and van Doorn [16] (which can be used for the same sort of bootstrapping argument) albeit a somewhat modest one; in essence, the translation eliminates clutter in the system that does not affect normalization. This work is done in the framework of *tiered* pure type systems, a simple class of persistent systems which is sufficient to study when concerned with questions about normalization.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases pure type systems, normalization, reduction-path-preserving translations, Barendregt-Geuvers-Klop conjecture

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.7

Acknowledgements I'd like to thank Stuart Kurtz for many useful discussions as well as the reviewers for their incredibly helpful comments.

1 Introduction

The class of pure type systems [2, 3, 5, 8, 9, 18] was introduced as a natural generalization of the lambda cube which includes systems with more complex sort structure and product type formation rules. The study of pure type systems is primarily concerned with how this sort structure affects meta-theoretic properties (especially given the minimal collection of type formers). One such property is normalization: a type system is *weakly normalizing* if every typable term has a normal form and *strongly normalizing* if no typable term appears in an infinite reduction sequence.

Despite the fact that weak normalization is, of course, the weaker of the two properties, it is often sufficient for proving other important meta-theoretic properties, e.g., consistency and decidability of type checking in the presence of dependent types. Observations to this effect were made by Geuvers in his PhD thesis [9], where he also conjectured that weak normalization implies strong normalization for *all* pure type systems (Conjecture 8.1.12). This conjecture has come to be known as the *Barendregt-Geuvers-Klop conjecture.*[1]

Little progress has been made on this conjecture, in part because pure type systems in general are not always amenable to standard techniques. Though natural, the generalization to pure type systems from the lambda cube is in some sense the most obvious one, a

[1] Sørensen submitted the conjecture by this name to the *TLCA List of Open Problems* [1]. He has also referred to it as the *Barendregt-Geuvers conjecture* [17]. Barendregt is noted by Barthe et al. [4] to have presented the conjecture at the *Second International Conference on Typed Lambda Calculi and Applications (1995)*, and Klop is the co-author of a preprint which refers to the Barendregt-Geuvers-Klop conjecture by name ([13], Conjecture 1.1).

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 7; pp. 7:1–7:21
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

basic syntactic ambiguation of the inference rules which allows for maximal freedom in sort structure. The resulting systems may fail to have the meta-theoretic properties one might expect (e.g., type unicity) so it is common to consider classes of systems which do maintain these properties. The state of the art of the conjecture is the result of Barthe et al. [4], which proves strong normalization from weak normalization for a class of non-dependent pure type systems (see Definition 11) by generalizing Xi's [19] and Sørensen's [17] CPS translation.

I propose revisiting the Barendregt-Geuvers-Klop conjecture in a slightly simpler framework. I begin by presenting a class of basic, concrete pure type systems I call *tiered* pure type systems. Despite their simplicity, they can be used to characterize a general class of pure type systems; so called *bounded separable persistent* pure type systems (and, in particular, bounded non-dependent systems) are disjoint unions of tiered systems.

Being concrete, the conjecture restricted to this setting is that *weak normalization implies strong normalization for all* ***tiered*** *pure type systems.* This re-framing of the problem is a minor though I believe important step towards making further progress on the full version of the conjecture. But even in this setting, there are many systems to consider, some of which contain what amounts to "junk" structure. The primary contribution of this paper is a translation of pure type systems which preserves typability and infinite reduction paths (I will simply write "path-preserving" from this point forward) and removes some of this irrelevant structure. By "removing structure" here, I mean that the target system of the translation is the same as the source system but with some sorts and rules removed.

Consider, for example, the system $\lambda\mathsf{HOL}$, which may be thought of as the system $\lambda\omega$ with an additional *superkind* sort $\triangle$ which allows for the introduction of kind variables that can appear in expressions but cannot be abstracted over. In $\lambda\mathsf{HOL}$, it is possible to derive

$$\mathfrak{A} : \square \vdash_{\lambda\mathsf{HOL}} \lambda A^{\mathfrak{A}}.\ \lambda x^{A}.\ x : \Pi A^{\mathfrak{A}}.\ A \to A.$$

A judgment of this form cannot derived in $\lambda\omega$ because the variable $\mathfrak{A}$ cannot be introduced without the axiom $\vdash_{\lambda\mathsf{HOL}} \square : \triangle$. Thus, the introduction of $\triangle$ is meaningful with respect to what expressions can be derived. But both $\lambda\mathsf{HOL}$ and $\lambda\omega$ are strongly normalizing. One basic observation is that there is a single expression inhabiting $\triangle$, namely $\square$. This sparsity of inhabitation can be leveraged to define a path-preserving translation from $\lambda\mathsf{HOL}$ to $\lambda\omega$ and, in fact, from any pure type system with an isolated top-sort to the same system but without the top-sort. In the case of the judgment above, the variable $\mathfrak{A}$ can be instantiated at $*$ yielding the judgment

$$\vdash_{\lambda\omega} \lambda A^{*}.\ \lambda x^{A}.\ x : \Pi A^{*}.\ A \to A$$

derived which can be derived in $\lambda\omega$.

I generalize this observation in two ways. First, I define a path-preserving translation that eliminates not just top-sorts but also any sort which is *top-sort-like*. Second, I extend this translation to eliminate not just isolated sorts, but also sorts which may appear in some rules. This translation can be iteratively applied to $\lambda\mathcal{S}$ until a fixed point $\lambda\mathcal{S}^{\downarrow}$ is reached. Thus, it can be used to prove the strong normalization of systems $\lambda\mathcal{S}$ for which $\lambda\mathcal{S}^{\downarrow}$ is known to be strongly normalizing. It can also be bootstrapped with existing results for the Barendregt-Geuvers-Klop conjecture. The argument is simple: if $\lambda\mathcal{S}$ is weakly normalizing, then so is $\lambda\mathcal{S}^{\downarrow}$ since it can be embedded in $\lambda\mathcal{S}$. By assumption, $\lambda\mathcal{S}^{\downarrow}$ is strongly normalizing, and so $\lambda\mathcal{S}$ is strongly normalizing by the path-preserving translation. Bootstrapping with the result of Barthe et al. yields a proof of the Barendregt-Geuvers-Klop conjecture for a larger class of systems. In particular, on a technical note, $\lambda\mathcal{S}$ may have *dependent* rules and *non-negatable* sorts (see ([4], Definition 2.23, Definition 3.1) and Definition 11 for details).

This technique bears a resemblance to the one used by Roux and van Doorn [16] in their structural theory of pure type systems, which in turn resembles the techniques of Geuvers and Nederhof [8] and Harper et al. [10]. In all these works, a translation is defined from one pure type system into another which has fewer rules. And though it is not explicitly stated, the translation of Roux and van Doorn can be bootstrapped in the same way as described above. In fact, their translation can be used to eliminate some rules *between* tiered systems in a disjoint union whereas the translation presented here eliminates some rules *within* the individual summands in a disjoint union of tiered systems (all while preserving strong normalization).

It is important to emphasize that this result depends on the fact that the additional structure that can be handled is irrelevant and, in particular, irrelevant with respect to normalization, not derivability or expressibility. But if we do want to prove the full conjecture, we also have to prove it for "junk" systems, ones which may not be interesting in their own right and may have rules which don't add much to the system. This result is perhaps more meaningfully interpreted in the reverse direction: the systems $\lambda\mathcal{S}^{\downarrow}$ for which the conjecture is *not* known to hold are targets for the developments of better techniques. Ideally, some technique could handle all these systems uniformly, but as of now it may be useful to further develop the theory regarding what barriers exist, and what systems beyond the lambda cube – natural or not – may be important to study.

In what follows I present some preliminary material, which includes some exposition on tiered systems. I then define the irrelevancy-eliminating translation in two parts: one part for eliminating rules and one for eliminating sorts. The final translation will be taken as the composition of these two translations. Finally, I present its application to the Barendregt-Geuvers-Klop conjecture and conclude with a short section on what it implies about the systems which remain to be studied.

2 Preliminaries

The class of pure type systems is the basis of a very general framework for describing type systems and their meta-theory. These systems vary in their sort structure and their product type formation rules, and include the entire lambda cube. Barendregt cites Berardi [5] and Terlouw [18] for their conception, though Geuvers and Nederhof [8] are cited as having given the first explicit definition, based on the previous two works. The presentations of Barendregt [2, 3] are perhaps the best known sources.[2]

A pure type system is specified by a triple of sets $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ satisfying $\mathcal{A} \subset \mathcal{S} \times \mathcal{S}$ and $\mathcal{R} \subset \mathcal{S} \times \mathcal{S} \times \mathcal{S}$. The elements of $\mathcal{S}$, $\mathcal{A}$, and $\mathcal{R}$ are called sorts, axioms and rules, respectively. I use s and t as meta-variables for sorts.[3]

For each sort s, fix a $\mathbb{Z}^+$-indexed set of expression variables V_s. Let ${}^{s}v_i$ denote the ith expression variable in V_s and let V denote $\bigcup_{s\in\mathcal{S}} \mathsf{V}_s$. I use x, y, and z as meta-variables for expression variables. The choice to annotate variables with sorts is one of convenience. The annotations can be dropped for the systems I consider, and are selectively included in the exposition. This observation regarding variable annotations was first made by Geuvers ([9], Definition 4.2.9).

[2] I am of the opinion that, after the development of the lambda cube, the notion of pure type systems was soon to follow, and that all aforementioned should be cited as originators.

[3] For any subsequent meta-variables, I use positive integer subscripts and tick marks, e.g., s_1, s_2, and s'. Note, however, that in later sections, s_i will refer to a particular sort in tiered systems. I will try to be as clear as possible when distinguishing between these two cases of notation.

The set of expressions of a pure type system with sorts $\mathcal{S}$ is described by the grammar

$$\mathsf{T} ::= \mathcal{S} \mid \mathsf{V} \mid \Pi \mathsf{V}^{\mathsf{T}}.\ \mathsf{T} \mid \lambda \mathsf{V}^{\mathsf{T}}.\ \mathsf{T} \mid \mathsf{TT}$$

I use capital modern English letters like M, N, P, Q, A, B, and C as meta-variables for expressions. Free variables, bound variables, α-congruence, β-reduction, substitution, sub-expressions, etc. are defined as usual (see, for example, Barendregt's presentation [3]). Substitution of x with N in M is denoted $M[N/x]$, and I write $N \subset M$ for "N is a sub-expression of M."

A **statement** is a pair of expressions, denoted $M : A$. The first expression is called the **subject** and the second is called the **predicate**. A **proto-context** is a sequence of statements whose subjects are expression variables. The statements appearing in proto-contexts are called **declarations**. I use capital Greek letters like Γ, Δ, Φ, and Υ as meta-variables for contexts. Often the sequence braces of contexts are dropped and concatenation of contexts is denoted by comma-separation. The β-equality relation and substitution extend to contexts element-wise. For a context Γ and declaration $(x : A)$ I write $(x : A) \in \Gamma$ if that declaration appears in Γ, and $\Gamma \subset \Delta$ if $(x : A) \in \Gamma$ implies $(x : A) \in \Delta$. A **proto-judgment** is a proto-context together with statement, denoted $\Gamma \vdash M : N$. The designation "judgment" is reserved for proto-judgments that are derivable according to the rules below. Likewise, the designation "context" is reserved for proto-contexts that appear in some (derivable) judgment.

The pure type system $\lambda\mathcal{S}$ specified by $(\mathcal{S}, \mathcal{A}, \mathcal{R})$ has the following rules for deriving judgments. In what follows, the meta-variables s and s' range over all sorts in $\mathcal{S}$ when unspecified. A variable ${}^{s}x$ is **fresh** with respect to a context Γ if it does not appear anywhere in Γ.

- **Axioms.** $\vdash_{\lambda\mathcal{S}} s : s'$ for any axiom (s, s').
- **Variable Introduction.** For a variable ${}^{s}x$ which is fresh with respect to Γ

$$\frac{\Gamma \vdash_{\lambda\mathcal{S}} A : s}{\Gamma, {}^{s}x : A \vdash_{\lambda\mathcal{S}} {}^{s}x : A}$$

- **Weakening.** For a variable ${}^{s}x$ which is fresh with respect to Γ

$$\frac{\Gamma \vdash_{\lambda\mathcal{S}} M : A \qquad \Gamma \vdash_{\lambda\mathcal{S}} B : s}{\Gamma, {}^{s}x : B \vdash_{\lambda\mathcal{S}} M : A}$$

- **Product Type Formation/Generalization.** For any rule (s, s', s'')

$$\frac{\Gamma \vdash_{\lambda\mathcal{S}} A : s \qquad \Gamma, {}^{s}x : A \vdash_{\lambda\mathcal{S}} B : s'}{\Gamma \vdash_{\lambda\mathcal{S}} \Pi {}^{s}x^{A}.\ B : s''}$$

- **Abstraction.**

$$\frac{\Gamma, {}^{s}x : A \vdash_{\lambda\mathcal{S}} M : B \qquad \Gamma \vdash_{\lambda\mathcal{S}} \Pi {}^{s}x^{A}.\ B : s'}{\Gamma \vdash_{\lambda\mathcal{S}} \lambda {}^{s}x^{A}.\ M : \Pi {}^{s}x^{A}.\ B}$$

- **Application.**

$$\frac{\Gamma \vdash_{\lambda\mathcal{S}} M : \Pi {}^{s}x^{A}.\ B \qquad \Gamma \vdash_{\lambda\mathcal{S}} N : A}{\Gamma \vdash_{\lambda\mathcal{S}} MN : B[N/{}^{s}x]}$$

- **Conversion.** For any terms A and B such that $A =_{\beta} B$

$$\frac{\Gamma \vdash_{\lambda\mathcal{S}} M : A \qquad \Gamma \vdash_{\lambda\mathcal{S}} B : s}{\Gamma \vdash_{\lambda\mathcal{S}} M : B}$$

The subscript on the turnstile is dropped when there is no fear of ambiguity. The annotations on bound variables in Π-expressions and λ-expressions are non-standard, and will in most cases be dropped, but they are occasionally useful to maintain (e.g., see Lemma 1). It is also standard to write $A \to B$ for $\Pi x^A.\ B$ in the case that x does not appear free in B.

An expression M is said to be **derivable** in $\lambda\mathcal{S}$ if there is some context Γ and expression A such that $\Gamma \vdash_{\lambda\mathcal{S}} M : A$. Although there is no distinction between terms and types, it is useful to call a judgment a **type judgment** if it is of the form $\Gamma \vdash A : s$ where $s \in \mathcal{S}$, and a **term judgment** if it is of the form $\Gamma \vdash M : A$ where $\Gamma \vdash A : s$ for some sort s. I also write that M is a term and A is a type in this case. By type correctness (Lemma 2), a judgment that is not a type judgment is a term judgment, though some judgments are both type and term judgments.

2.1 Meta-Theory

I collect here the meta-theoretic lemmas necessary for the subsequent results. Much of the meta-theory of pure type systems was worked out by Geuvers and Nederhof [8], and can be found in several of the great available resources on pure type systems ([3, 4, 9, 12], among others) so proofs are omitted. For the remainder of the section, fix a pure type system $\lambda\mathcal{S}$.

▶ **Lemma 1** (Generation). *For any context Γ and expression A, the following hold.*

- *Sort. For any sort s, if $\Gamma \vdash s : A$, then there is a sort s' such that $A =_\beta s'$ and $(s, s') \in \mathcal{A}$.*
- *Variable. For any sort s and variable ${}^sx$, if $\Gamma \vdash {}^sx : A$, then there is an type B such that $\Gamma \vdash B : s$ and $({}^sx : B)$ appears in Γ and $A =_\beta B$.*
- *Π-expression. For any sort s and expressions B and C, if $\Gamma \vdash \Pi^s x^B.\ C : A$ then there are sorts s', and s'' such that $\Gamma \vdash B : s$ and $\Gamma, {}^sx : B \vdash C : s'$ and $(s, s', s'') \in \mathcal{R}$ and $A =_\beta s''$.*
- *λ-expression. For any sort s and expressions B and M, if $\Gamma \vdash \lambda^s x^B.\ M : A$ then there is a type C and sort s' such that such that $\Gamma \vdash \Pi^s x^B.\ C : s'$ and $\Gamma, {}^sx : B \vdash M : C$ and $A =_\beta \Pi^s x^B.\ C$.*
- *Application. For expressions M and N, if $\Gamma \vdash MN : A$, then there is a sort s and types B and C such that $\Gamma \vdash M : \Pi^s x^B.\ C$ and $\Gamma \vdash N : B$ and $A =_\beta C[N/{}^sx]$.*

▶ **Lemma 2** (Type Correctness). *For any context Γ and expressions M and A, if $\Gamma \vdash M : A$ then $A \in \mathcal{S}$ or there is a sort s such that $\Gamma \vdash A : s$.*

▶ **Definition 3.** *A pure type system is **functional** if the following hold.*

- *If $(s, t) \in \mathcal{A}$ and $(s, t') \in \mathcal{A}$ then $t = t'$.*
- *If $(s, t, u) \in \mathcal{R}$ and $(s, t, u') \in \mathcal{R}$, then $u = u'$.*

▶ **Lemma 4** (Type Unicity). *If $\lambda\mathcal{S}$ is functional then for any context Γ and expressions M, A, and B, if $\Gamma \vdash M : A$ and $\Gamma \vdash M : B$, then $A =_\beta B$.*

▶ **Definition 5.** *A sort s is a **top-sort** if there is no sort s' such that $(s, s') \in \mathcal{A}$. It is a **bottom-sort** if there is no sort s' such that $(s', s) \in \mathcal{A}$.*

▶ **Lemma 6** (Top-Sort Lemma). *For any context Γ, variable x, expressions A and B, and top-sort s the following hold.*

1. $\Gamma \nvdash s : A$
2. $\Gamma \nvdash x : s$
3. $\Gamma \nvdash AB : s$

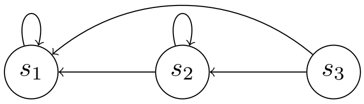


Figure 1 A visual representation of the system λU.

2.2 Tiered Pure Type Systems

General pure type systems are notoriously difficult to work with so it is typical to consider classes of pure type systems satisfying certain properties, e.g., persistence as subsequently defined.

▶ **Definition 7.** *A pure type system $\lambda\mathcal{S}$ is* ***persistent*** *if it is functional (Definition 3) and*

- *if $(s,t) \in \mathcal{A}$ and $(s',t) \in \mathcal{A}$ then $s = s'$;*
- $\mathcal{R}_{\lambda\mathcal{S}} \subset \{(s,s',s') \mid (s,s') \in \mathcal{S} \times \mathcal{S}\}$.

From this point forward, I freely use the notation (s,s') for the rule (s,s',s'). One minor issue with properties like this is that it is often difficult to envisage the systems which satisfy them. In particular, the results tailored to a class of systems defined as such may use more meta-theoretic machinery than necessary. I choose, instead, to work with a simple class of systems I call *tiered* pure type systems, which have a very concrete description.

▶ **Definition 8.** *Let n be a non-negative integer. A pure type system is* ***n-tiered*** *if its has the form*

$$\mathcal{S} = \{s_i \mid i \in [n]\}$$
$$\mathcal{A} = \{(s_i, s_{i+1}) \mid i \in [n-1]\}$$
$$\mathcal{R} \subset \{(s,s',s') \mid (s,s') \in \mathcal{S} \times \mathcal{S}\}$$

where $[n] \triangleq \{1,\ldots,n\}$.

A couple remarks about these systems:

- these systems can be envisaged as graphs as in Figure 1, which is a visual representation of the 3-tiered system λU. In such representations, an arrow (s_i,s_j) indicates the presence of the rule (s_i,s_j). Axioms are not represented in the graph except in the ordered the nodes are presented;
- the only 0-tiered system is the empty pure type system; there are two 1-tiered systems, specified by either $(\{s_1\},\emptyset,\emptyset)$ or $(\{s_1\},\emptyset,\{(s_1,s_1)\})$, neither of which have derivable expressions; the 2-tiered systems which contain the rule (s_1,s_1) are exactly the lambda cube;
- the n-tiered systems are considered in passing by Barthe et al. ([4], Remark 2.39). They include natural subsystems of ECC^n (as defined in [15]) with only the two-sorted rules;

A large class of natural pure type systems can be classified as disjoint unions of tiered systems. In order to state this equivalence, I work in the structural theory of pure type systems presented by Roux and van Doorn [16].

▶ **Definition 9.** *For pure type systems $\lambda\mathcal{S}$ and $\lambda\mathcal{S}'$, the* ***disjoint union*** *$\lambda\mathcal{S} \sqcup \lambda\mathcal{S}'$ is specified by*

$$\mathcal{S}_{\lambda\mathcal{S}\sqcup\lambda\mathcal{S}'} \triangleq \mathcal{S}_{\lambda\mathcal{S}} \sqcup \mathcal{S}_{\lambda\mathcal{S}'}$$
$$\mathcal{A}_{\lambda\mathcal{S}\sqcup\lambda\mathcal{S}'} \triangleq \mathcal{A}_{\lambda\mathcal{S}} \cup \mathcal{A}_{\lambda\mathcal{S}'}$$
$$\mathcal{R}_{\lambda\mathcal{S}\sqcup\lambda\mathcal{S}'} \triangleq \mathcal{R}_{\lambda\mathcal{S}} \cup \mathcal{R}_{\lambda\mathcal{S}'}$$

A class of systems which can be characterized by disjoint unions must be partitionable into atoms which can be analyzed individually. Let '$<_\mathcal{A}$', '$\leq_\mathcal{A}$', and '$\approx_\mathcal{A}$' denote the transitive, reflexive-transitive, and equivalence closure of $\mathcal{A}$, respectively.

▶ **Definition 10.** *A pure type system $\lambda\mathcal{S}$ is* ***separable*** *if $(s, s') \in \mathcal{R}_{\lambda\mathcal{S}}$ implies $s \approx_\mathcal{A} s'$. It is* ***atomic*** *if $s \approx_\mathcal{A} s'$ for all sorts s and s'.*

There are, of course, many examples of important *non-separable* persistent pure type systems, e.g., systems from the logic cube [2, 3, 6, 9, 7] like Berardi's formulation of λPREDω which is specified by

$$\begin{aligned}
\mathcal{S} &\triangleq \{*^s, \Box^s, *^p, \Box^p\}\\
\mathcal{A} &\triangleq \{(*^s, \Box^s), (*^p, \Box^p)\}\\
\mathcal{R} &\triangleq \{(*^p, *^p), (\Box^p, *^p), (\Box^p, \Box^p), (*^s, *^p), (*^s, \Box^p)\}
\end{aligned}$$

The rules $(*^s, *^p)$ and $(*^s, \Box^p)$ "cross" between two tiered systems.[4] Despite this, there are also useful classes of systems which *are* separable, e.g., generalized non-dependent systems are separable by fiat.

▶ **Definition 11.** *Let $\lambda\mathcal{S}$ be a pure type system.*

- *$\lambda\mathcal{S}$ satisfies the* ***ascending chain condition*** *if '$<_\mathcal{A}$' does, i.e., there is no infinite sequence of sorts $s, s', s'', \ldots$ such that $s < s' < s'' \ldots$; it satisfies the* ***descending chain condition*** *if there is no infinite sequence of sorts $s, s', s'', \ldots$ such that $s > s' > s'' \ldots$; it is* ***bounded*** *if it satisfies both the ascending and descending chain conditions.*
- *$\lambda\mathcal{S}$ is* ***weakly non-dependent*** *if $(s, s', s'') \in \mathcal{R}$ implies $s \geq s' \geq s''$.*
- *$\lambda\mathcal{S}$ is* ***stratified*** *if it satisfies the ascending chain condition and is non-dependent.*
- *$\lambda\mathcal{S}$ is* ***generalized non-dependent*** *if it is stratified and persistent. If $\lambda\mathcal{S}$ is also bounded, I will write that it is* ***bounded non-dependent****, and if it is tiered, I will just write that it is non-dependent.*

We can now characterize disjoint unions of tiered systems in terms of the above properties. The proof of Lemma 12 is omitted, but it follows roughly by showing that '$\leq_\mathcal{A}$' is a total order.

▶ **Lemma 12.** *A pure type system is tiered if and only if it is persistent, bounded, and atomic.*

▶ **Lemma 13.** *A pure type system is persistent, bounded, and separable if and only if is the disjoint union of tiered pure type systems.*

Proof. It is straightforward to verify that tiered systems are persistent, bounded, and atomic, and so their disjoint unions are persistent, bounded, and separable. In the other direction, let $\lambda\mathcal{S}$ be a pure type system that is persistent, bounded, and separable and consider the partition $\mathcal{P}$ of $\mathcal{S}$ into $\approx_\mathcal{A}$-equivalence classes. Let $\mathcal{S}_p$ be such an equivalence class and let $\lambda\mathcal{S}_p$ denote the pure system specified by

$$\begin{aligned}
\mathcal{S}_{\lambda\mathcal{S}_p} &\triangleq \mathcal{S}_p\\
\mathcal{A}_{\lambda\mathcal{S}_p} &\triangleq \mathcal{A}_{\lambda\mathcal{S}} \cap (\mathcal{S}_p \times \mathcal{S}_p)\\
\mathcal{R}_{\lambda\mathcal{S}_p} &\triangleq \mathcal{R}_{\lambda\mathcal{S}} \cap (\mathcal{S}_p \times \mathcal{S}_p \times \mathcal{S}_p)
\end{aligned}$$

[4] I'd like to specifically thank the reviewer who reminded me of this example.

The system $\lambda\mathcal{S}_p$ is persistent and bounded because $\lambda\mathcal{S}$ is, and it is atomic by definition, so by Lemma 12 it is tiered. We can then view $\lambda\mathcal{S}$ as the system $\bigsqcup_{\mathcal{S}_p \in \mathcal{P}} \lambda\mathcal{S}_p$.[5] Note that all axioms are accounted for by fiat and all rules are accounted for by separability. ◀

▶ **Corollary 14.** *A pure type system is bounded non-dependent if and only if it is the disjoint union of non-dependent tiered pure types systems.*

Roux and van Doorn [16] show that the (strong) normalization of a disjoint union of pure type systems is equivalent to the (strong) normalization of each of its individual summands. So on questions of normalization regarding persistent, bounded, separable systems it suffices to consider tiered systems.

▶ **Proposition 15.** *If weak normalization implies strong normalization for all tiered pure type systems, then the same is true for all persistent, bounded, separable pure type systems. In particular, if weak normalization implies strong normalization for all non-dependent pure type systems, then the same is true for all bounded non-dependent pure type systems.*[6]

I close this section with some useful features of tiered systems. One of the primary benefits of working in persistent systems in general (and tiered systems in particular) is that derivable expressions can be classified by the *level* in the system at which they are derivable. This property is shown by defining a degree measure on expressions and classifying expressions according to their degree. This result is due to Berardi [6] and Geuvers and Nederhof [8], and the presentation here roughly follows the same course.

▶ **Definition 16.** *The* **degree** *of an expression is given by the following function* $\deg : \mathsf{T} \to \mathbb{N}$.

$$\begin{aligned} \deg(s_i) &\triangleq i + 1 \\ \deg({}^{s_i}x) &\triangleq i - 1 \\ \deg(\Pi x^A.\ B) &\triangleq \deg(B) \\ \deg(\lambda x^A.\ M) &\triangleq \deg(M) \\ \deg(MN) &\triangleq \deg(M) \end{aligned}$$

Let T_j *denote* $\{M \in \mathsf{T} \mid \deg(M) = j\}$ *and let* $\mathsf{T}_{\geq j}$ *denote* $\{M \in \mathsf{T} \mid \deg(M) \geq j\}$.

▶ **Lemma 17** (Classification). *Let* $\lambda\mathcal{S}$ *be an* n*-tiered pure type system. For any expression* A*, the following hold.*

- $\deg(A) = n + 1$ *if and only if* $A = s_n$.
- $\deg(A) = n$ *if and only if* $\Gamma \vdash_{\lambda\mathcal{S}} A : s_n$ *for some context* Γ.
- *For* $i \in [n-1]$, *we have* $\deg(A) = i$ *if and only if* $\Gamma \vdash_{\lambda\mathcal{S}} A : B$ *and* $\Gamma \vdash_{\lambda\mathcal{S}} B : s_{i+1}$ *for some context* Γ *and expression* B.

In particular, for context Γ *and expressions* M *and* A*, if* $\Gamma \vdash_{\lambda\mathcal{S}} M : A$ *then* $\deg(A) = \deg(M) + 1$.

[5] Formally, they are isomorphic pure type systems. The definition of a pure type system homomorphism is as one might expect, see the definition of Geuvers ([9], Definition 4.2.5) – which is also used by Roux and van Doorn [16] – for more details.

[6] This all sits in a more general theory. A natural extension of tiered systems includes infinite tiered systems and even cyclic systems, which can help better characterize classes of systems, like generalized non-dependent systems and persistent separable systems.

Finally, some useful facts about degree. See the presentation by Barendregt [3] for proofs in the 2-tiered case.

▶ **Lemma 18.** *Let $\lambda\mathcal{S}$ be an n-tiered pure type system and let A and B be expressions derivable in $\lambda\mathcal{S}$.*

- *If* $\deg(B) = j - 1$ *then* $\deg(A[B/^{s_j}x]) = \deg(A)$.
- *If* $A \twoheadrightarrow_\beta B$*, then* $\deg(A) = \deg(B)$.

3 Irrelevancy-Eliminating Translation

Fix an n-tiered pure type system $\lambda\mathcal{S}$. I first describe the sorts which are *top-sort-like*. Recall that s is a top-sort if there is no sort s' such that $(s, s') \in \mathcal{A}$, so s_n is the only top-sort of $\lambda\mathcal{S}$. Top-sorts are interesting in part because they tend to be sparsely inhabited. A top-sort-like sort s_i which is not a top-sort has the sort s_{i+1} above it, but to ensure s_i is sparsely inhabited, s_{i+1} should not appear in any rules. We will also be interested in top-sort-like sorts which themselves do not appear in any rules.

▶ **Definition 19.**

- *A sort s_i is **rule-isolated** if for all j, neither (s_j, s_i) nor (s_i, s_j) appear in $\mathcal{R}_{\lambda\mathcal{S}}$.*
- *A sort s_i is **top-sort-like** if $i < n$ implies s_{i+1} is rule-isolated (i.e., s_i is a top-sort or its succeeding sort is rule-isolated).*
- *A sort s_i is **completely isolated** if it is top-sort-like and rule-isolated.*

Next, I describe the structure that will be considered irrelevant with respect to normalization. Roughly speaking, this includes rules on top-sort-like sorts which allow for the derivation of redexes on expressions from sparsely inhabited types. It will be possible to essentially pre-reduce these redexes in the translation, eliminating the need for the rules in the target system of the translation. In what follows, it will be convenient to consider *sets* of top-sort-like sorts. I call a subset $\mathcal{I}$ of $[n]$ an *index set* for $\lambda\mathcal{S}$, and denote by $\mathcal{S}_\mathcal{I}$ the set $\{s_i \mid i \in \mathcal{I}\}$.

▶ **Definition 20.**

- *For any index set $\mathcal{J}$, a sort s_i is **$\mathcal{J}$-irrelevant** if there is no sort s_j such that $j \in \mathcal{J}$ and $(s_j, s_i) \in \mathcal{R}_{\lambda\mathcal{S}}$. A sort s_i is **irrelevant** if it is $[n]$-irrelevant.*
- *An index set $\mathcal{I}$ is **completely irrelevant** in $\lambda\mathcal{S}$, if for each i in $\mathcal{I}$,*
  - *s_i is top-sort-like and irrelevant;*
  - *s_{i-1} is $([n] \setminus \mathcal{I})$-irrelevant.*

In the case of complete irrelevance, if $\mathcal{I}$ is a singleton set $\{i\}$, then the only rule with s_{i-1} appearing second is (s_i, s_{i-1}). By considering sets of indices simultaneously, we can make weaker assumptions on these preceding sorts. The condition of $([n] \setminus \mathcal{I})$-irrelevance ensures that s_{i-1} becomes irrelevant after removing the rules associated with sorts in $\mathcal{S}_\mathcal{I}$. Note also that if $(s_i, s_i) \in \mathcal{R}_{\lambda\mathcal{S}}$, then any completely irrelevant index set cannot contain $i - 1$, i or $i + 1$. Finally, it is important that there is a *unique maximum completely irrelevant index set.* In particular, the union of any two completely irrelevant index sets is completely irrelevant.

3.1 Eliminating Completely Irrelevant Rules

This section contains the translation which removes the rules associated with sorts whose indices appear in a completely irrelevant index set. For the remainder of the section, fix such a set $\mathcal{I}$. We begin by showing that sorts in $\mathcal{S}_\mathcal{I}$ are sparsely inhabited.

▶ **Lemma 21.** *Let s_i be an irrelevant sort such that s_i is a top-sort or s_{i+1} is irrelevant. For every derivable expression A, if $\deg(A) = i$ then $A = s_{i-1}$ or $A \in \mathsf{V}_{s_{i+1}}$.*

Proof. If $i = n$, then this follows directly from the top-sort lemma (Lemma 6) and the fact that s_n is irrelevant. In fact, in this case s_n is inhabited solely by s_{n-1}. If $i \neq n$, this follows in a similar way, i.e., by induction on the structure of derivations. The cases in which the last inference is an axiom, variable introduction, weakening, or conversion are straightforward. The last inference cannot be a product type formation because s_i is irrelevant. The last inference cannot be an abstraction or application because s_{i+1} is irrelevant. ◀

This does not hold if s_{i+1} is not irrelevant. If $(s_{i+1}, s_{i+1}) \in \mathcal{R}_{\lambda\mathcal{S}}$, for example, then $\varnothing \vdash (\lambda x^{s_i}.\ x)s_{i-1} : s_i$ is derivable. This is why we require *both* s_i and s_{i+1} to be irrelevant.

The primary challenge moving forward is dealing with the fact that variables may appear as types of sort s_i. These variables are what will necessitate s_{i+1} being not just irrelevant, but also isolated. Regardless, the sparsity of types of sort s_i induces sparsity of expressions of degree $i - 1$.

▶ **Lemma 22.** *For index i in $\mathcal{I}$, context Γ and expression M, if $\Gamma \vdash M : s_{i-1}$, then M is of the form $\Pi x_1{}^{A_1}. \ldots \Pi x_k{}^{A_k}.\ B$ where $\deg(A_j) \in \mathcal{I}$ for all j and either $B = s_{i-2}$ or $B \in \mathsf{V}_{s_i}$.*

Proof. By induction on the structure of derivations. The cases in which the last inference is an axiom, variable introduction, or weakening are straightforward. The last inference clearly cannot be an abstraction, and it cannot be an application since s_i is irrelevant. What follows are the remaining two cases.

Product Type Formation. Suppose the last inference is of the form

$$\frac{\Gamma \vdash A : s_j \qquad \Gamma, x : A \vdash B : s_{i-1}}{\Gamma \vdash \Pi x^A.\ B : s_{i-1}}$$

Since s_{i-1} is $(\mathcal{S}_{\lambda\mathcal{S}} \setminus \mathcal{I})$-irrelevant, it must be that $j \in \mathcal{I}$. The desired result holds after applying the inductive hypothesis to the right antecedent judgment.

Conversion. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash s_{i-1} : s_i}{\Gamma \vdash M : s_{i-1}}$$

where $A =_\beta s_{i-1}$. Note that $\deg(A) = i$ so $\Gamma \vdash A : s_i$ by type correctness. Thus, $A = s_{i-1}$ by Lemma 21, which means the inductive hypothesis can be applied directly to the left antecedent judgment. ◀

▶ **Lemma 23.** *For index i in $\mathcal{I}$, context Γ, expression A and variable ${}^{s_{i+1}}x$, if $\Gamma \vdash A : {}^{s_{i+1}}x$, then $A \in \mathsf{V}_{s_i}$.*

Proof. By induction on the structure of derivations. The cases in which the last inference is an axiom, variable introduction, or weakening are straightforward. The last inference clearly cannot be a product type formation or an abstraction. The last inference cannot be an application because s_i is irrelevant. Finally, all conversions are trivial by the same argument as in the previous lemma. ◀

▶ **Corollary 24.** *For index i in $\mathcal{I}$, every derivable expression M of degree $i - 1$ is of the form $\Pi x_1{}^{A_1}. \ldots \Pi x_k{}^{A_k}.\ B$ where $\deg(A_j) \in \mathcal{I}$ for all j and $B = s_{i-2}$ or $B \in \mathsf{V}_{s_i}$ (and k may be 0).*

The Translation

The following translation is defined such that it essentially pre-reduces all redexes whose source types have degree in $\mathcal{I}$. Naturally, this means it does not strictly preserve β-reductions, but because these sources types are so sparsely inhabited, we can define a complexity measure on expressions which is monotonically decreasing in the β-reductions that are pre-performed by the translation. This is similar to the technique used by Sørensen for simulating π-reductions [17].

The other wrinkle in defining this translation is that it is difficult to pre-reduce expressions of variable type because even though such types are sparsely inhabited, it is unclear *a priori* what the value of the expression will be after a series of reductions. By Lemma 23, we know it reduces to a variable, but we don't know *which* variable, and it may be one that is generalized or abstracted over. We ensure this doesn't happen by requiring s_{i+1} is isolated, not just irrelevant. We also introduce a distinguished variable $\bullet_z$ of type z for each variable z of sort s_{i+1} in the context. This gives us a canonical term that the translation can assign to expressions of this type.

▶ **Definition 25.** *Define the context-indexed function* $\tau_\Gamma : \mathsf{Ctx} \times \mathsf{T} \to \mathsf{T}$ *by induction on both arguments as follows.*

$$\tau_\Gamma(s_i) \triangleq s_i$$

$$\tau_\Gamma({}^{s_i}x) \triangleq \begin{cases} s_{i-2} & \textit{if } i \in \mathcal{I} \textit{ and } ({}^{s_i}x : s_{i-1}) \in \Gamma \\ \bullet_z & \textit{if } i \in \mathcal{I} \textit{ and } ({}^{s_i}x : {}^{s_{i+1}}z) \in \Gamma \\ {}^{s_i}x & \textit{otherwise} \end{cases}$$

$$\tau_\Gamma(\Pi x^A.\ B) \triangleq \begin{cases} \tau_{\Gamma,x:A}(B) & \deg(A) \in \mathcal{I} \\ \Pi x^{\tau_\Gamma(A)}.\ \tau_{\Gamma,x:A}(B) & \textit{otherwise} \end{cases}$$

$$\tau_\Gamma(\lambda x^A.\ M) \triangleq \begin{cases} \tau_{\Gamma,x:A}(M) & \deg(A) \in \mathcal{I} \\ \lambda x^{\tau_\Gamma(A)}.\ \tau_{\Gamma,x:A}(M) & \textit{otherwise} \end{cases}$$

$$\tau_\Gamma(MN) \triangleq \begin{cases} \tau_\Gamma(M) & \deg(N) + 1 \in \mathcal{I} \\ \tau_\Gamma(M)\tau_\Gamma(N) & \textit{otherwise} \end{cases}$$

where $\bullet_z$ *is a distinguished variable. This function is used to define a function on contexts as*

$$\tau(\varnothing) \triangleq \varnothing$$

$$\tau(\Gamma, {}^{s_j}x : A) \triangleq \begin{cases} \tau(\Gamma) & j \in \mathcal{I} \\ \tau(\Gamma), {}^{s_j}x : s_{j-1}, \bullet_x : {}^{s_j}x & \textit{if } j-1 \in \mathcal{I} \textit{ and } A = s_{j-1} \\ \tau(\Gamma), {}^{s_j}x : \tau_\Gamma(A) & \textit{otherwise.} \end{cases}$$

As for proving the desired features of this translation, first note if $i \in \mathcal{I}$, then the translation maps expressions of degree $i-1$ (where $i \in \mathcal{I}$) to a sort or a $\bullet$-variable.

▶ **Proposition 26.** *For any index* i *in* $\mathcal{I}$*, context* Γ*, and term* A*, if* $\Gamma \vdash A : s_{i-1}$*, then* $\tau_\Gamma(A) = s_{i-2}$*, and if* $\Gamma \vdash A : {}^{s_{i+1}}x$ *for some variable* ${}^{s_{i+1}}x$*, then* $\tau_\Gamma(A) = \bullet_x$*.*

It suffices to consider the expressions of the form specified by Corollary 24, for which the above fact clearly holds. This turns out to be a key feature of the translation. Because the translation is able to drop so much information about these expressions, we can pre-reduce redexes in which they appear on the right.

We also use the fact that the context argument of the translation can be weakened when the last variable does not appear in the expression argument.

▶ **Proposition 27.** *For any context Γ, expressions M, A, and B, and variable x, if $\Gamma \vdash M : A$ and $\Gamma \vdash B : s_i$ then $\tau_{\Gamma, x:B}(M) = \tau_\Gamma(M)$.*

We now prove the standard substitution-commutation and β-preservation lemmas for this translation.

▶ **Lemma 28.** *For any index i, context Γ, expressions M, N, A and B, and variable ${}^{s_i}x$, if $\Gamma, {}^{s_i}x : A \vdash M : B$ and $\Gamma \vdash N : A$ then*

$$\tau_\Gamma(M[N/{}^{s_i}x]) = \begin{cases} \tau_{\Gamma, {}^{s_i}x:A}(M) & i \in \mathcal{I} \\ \tau_{\Gamma, {}^{s_i}x:A}(M)[\tau_\Gamma(N)/{}^{s_i}x] & \textit{otherwise.} \end{cases}$$

Proof. By induction on the structure of M. First suppose that $i \in \mathcal{I}$.

Sort. If M is of the form s_j, then $\tau_\Gamma(s_j[N/{}^{s_i}x]) = \tau_\Gamma(s_j)$.

Variable. First suppose M is of the form ${}^{s_i}x$. In particular, $A =_\beta B$, and since $\deg(A) = \deg(B) = i$, we have $A = B$ by Lemma 21. If $A = s_{i-1}$, then by Proposition 26 we have $\tau_\Gamma(N) = s_{i-2}$ and

$$\tau_\Gamma({}^{s_i}x[N/{}^{s_i}x]) = \tau_\Gamma(N) = s_{i-2} = \tau_{\Gamma, x:s_{i-1}}({}^{s_i}x).$$

Similarly, if A is of the form ${}^{s_{i+1}}y$, then $\tau_\Gamma(N) = \bullet_y$ and

$$\tau_\Gamma({}^{s_i}x[N/{}^{s_i}x]) = \tau_\Gamma(N) = \bullet_y = \tau_{\Gamma, x:y}({}^{s_i}x).$$

If M is of the form ${}^{s_j}y$ where ${}^{s_j}y \neq {}^{s_i}x$, then $\tau({}^{s_j}y[N/{}^{s_i}x]) = \tau({}^{s_j}y)$.

Π-Expression. If M is of the form $\Pi y^A.\ B$, then

$$\begin{aligned} \tau_\Gamma((\Pi y^A.\ B)[N/x]) &= \tau_\Gamma(\Pi y^{A[N/x]}.\ B[N/x]) \\ &= \begin{cases} \tau_{\Gamma, y:A}(B) & \deg(A) \in \mathcal{I} \\ \Pi y^{\tau_\Gamma(A)}.\ \tau_{\Gamma, y:A}(B) & \text{otherwise} \end{cases} \end{aligned}$$

where the last equality follows from the definition of τ and the inductive hypothesis. This also depends on Proposition 27 to show that $\tau_{\Gamma, y:A}(A) = \tau_\Gamma(A)$. The cases in which M is a λ-expression or application are similar. Furthermore, when $i \notin \mathcal{I}$, all cases are analogous. ◀

Before proving the β-preservation lemma, it is convenient to partition the β-reduction relation into two parts, one part which is directly preserved by the translation (β_1) and one part which is pre-reduced by the translation (β_2).

▶ **Definition 29.** *Let β_2 denote the notion of reduction given by*

$$(\lambda x^A.\ M)N \to_{\beta_2} M[N/x]$$

where $\deg(A) \in \mathcal{I}$, extended to a congruence relation in the usual way. Let β_1 denote the same notion of reduction but with $\deg(A) \notin I$, so that $\beta_1 \cap \beta_2 = \emptyset$ and $\beta_1 \cup \beta_2 = \beta$.

▶ **Lemma 30.** *For expressions M and N derivable in the context Γ, the following hold.*

- *If $M \to_{\beta_1} N$, then $\tau_\Gamma(M) \to_\beta \tau_\Gamma(N)$;*
- *if $M \to_{\beta_2} N$, then $\tau_\Gamma(M) = \tau_\Gamma(N)$;*
- *in particular, if $M =_\beta N$, then $\tau_\Gamma(M) =_\beta \tau_\Gamma(N)$.*

Proof. The last item follows directly from the first two. We prove the first two items by induction on the structure of the one-step β-reduction relation. In the case a redex $(\lambda x^A.\ M)N$, if $\deg(A) \notin \mathcal{I}$, then we have

$$\begin{aligned}\tau_\Gamma((\lambda x^A.\ M)N) &= \tau_\Gamma(\lambda x^A.\ M)\tau_\Gamma(N)\\ &= (\lambda x^{\tau_\Gamma(A)}.\ \tau_{\Gamma,x:A}(M))\tau_\Gamma(N)\\ &\to_\beta \tau_{\Gamma,x:A}(M)[\tau_\Gamma(N)/x]\\ &= \tau_\Gamma(M[N/x])\end{aligned}$$

and otherwise,

$$\begin{aligned}\tau_\Gamma((\lambda x^A.\ M)N) &= \tau_\Gamma(\lambda x^A.\ M)\\ &= \tau_{\Gamma,x:A}(M)\\ &= \tau_\Gamma(M[N/x])\end{aligned}$$

where the last equality in each sequence of equalities follows from the substitution-commutation lemma (Lemma 28). To show the desired result holds up to congruences, it must follow that expressions dropped by the translation are already in normal form.

Π-Expression. Suppose M is of the form $\Pi x^A.\ B$ and N is of the form $\Pi x^{A'}.\ B'$ where

$$\Pi x^A.\ B \to_\beta \Pi x^{A'}.\ B'$$

If $\deg(A) \notin \mathcal{I}$, then either $A \to_\beta A'$ and $B = B'$ or $B \to_\beta B'$ and $A = A'$ and the inductive hypothesis can be safely applied. If $\deg(A) \in \mathcal{I}$, then Lemma 21 implies that A is in normal form, so $A = A'$ and $B \to_\beta B'$, and the inductive hypothesis can be safely applied. The case in which M is a λ-expression is similar.

Application. Suppose M is of the form PQ and N is of the form $P'Q'$ where

$$PQ \to_{\beta_1} P'Q'$$

If $\deg(Q) + 1 \notin \mathcal{I}$, then either $P \to_\beta P'$ and $Q = Q'$ or $Q \to_\beta Q'$ and $P = P'$ and the inductive hypothesis can be safely applied. If $\deg(Q) + 1 \in \mathcal{I}$, Corollary 24 implies that Q is in normal form, so $Q = Q'$ and $P \to_\beta P'$ and the inductive hypothesis can be safely applied. ◀

With these two lemmas, we can now prove that the translation preserves typability. The system we translate to is defined simply as the one in which the rules associated with sorts in $\mathcal{S}_\mathcal{I}$ are dropped.

▶ **Definition 31.** *The* ***irrelevance reduction*** *of $\lambda\mathcal{S}$, denoted here by $\lambda\mathcal{S}^-$, is the n-tiered system specified by the rules $\mathcal{R}_{\lambda\mathcal{S}} \setminus \{(s_i, s_j) \mid i \in \mathcal{I}$ and $j \in [n]\}$.*

▶ **Lemma 32.** *For context Γ and expressions M and A, if*

$$\Gamma \vdash_{\lambda\mathcal{S}} M : A \qquad \textit{then} \qquad \tau(\Gamma) \vdash_{\lambda\mathcal{S}^-} \tau_\Gamma(M) : \tau_\Gamma(A).$$

Proof. By induction on the structure of derivations.

Axiom. If the derivation is a single axiom $\vdash s_i : s_{i+1}$ then the translated derivation is the same axiom.

Variable Introduction. Suppose the last inference is of the form

$$\frac{\Gamma \vdash A : s_i}{\Gamma, {}^{s_i}x : A \vdash {}^{s_i}x : A}$$

First suppose $i \in \mathcal{I}$. If $A = s_{i-1}$, then $\tau_{\Gamma, x:s_{i-1}}(x) = s_{i-2}$ and

$$\tau(\Gamma) \vdash s_{i-2} : s_{i-1}$$

where $\tau(\Gamma)$ is well-formed by the inductive hypothesis; that is,

$$\tau(\Gamma) \vdash \tau_\Gamma(A) : s_i$$

implies $\tau(\Gamma)$ is well-formed. If A is of the form ${}^{s_{i+1}}y$, then $({}^{s_{i+1}}y : s_{i-1}) \in \Gamma$, which implies $(\bullet_y : {}^{s_{i+1}}y) \in \tau(\Gamma)$ and $\tau(\Gamma) \vdash \bullet_y : {}^{s_{i+1}}y$ where $\tau(\Gamma)$ is again well-formed by the inductive hypothesis.

Next suppose $i - 1 \in \mathcal{I}$ and $A = s_{i-1}$. By the inductive hypothesis, we can derive

$$\frac{\tau(\Gamma) \vdash s_{i-1} : s_i}{\tau(\Gamma), {}^{s_i}x : s_{i-1} \vdash {}^{s_i}x : s_{i-1}}$$

and so by weakening,

$$\frac{\tau(\Gamma), {}^{s_i}x : s_{i-1} \vdash {}^{s_i}x : s_{i-1} \qquad \tau(\Gamma), {}^{s_i}x : s_{i-1} \vdash {}^{s_i}x : s_{i-1}}{\tau(\Gamma), {}^{s_i}x : s_{i-1}, \bullet_x : {}^{s_i}x \vdash {}^{s_i}x : s_{i-1}}$$

The remaining cases are straightforward.

Weakening. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s_i}{\Gamma, {}^{s_i}x : B \vdash M : A}$$

By Proposition 26, we have $\tau_{\Gamma, x:B}(M) = \tau_\Gamma(M)$. By type correctness, $\Gamma \vdash A : s_j$ for some index j, so $\tau_{\Gamma,x:B}(A) = \tau_\Gamma(A)$. So the inductive hypothesis implies

$$\tau(\Gamma) \vdash \tau_{\Gamma,x:B}(M) : \tau_{\Gamma,x:B}(A)$$

We can then use an argument similar to the one in the previous case to extend the context to $\tau(\Gamma, x : B)$.

Product Type Formation. Suppose the last inference is of the form

$$\frac{\Gamma \vdash A : s_i \qquad \Gamma, x : A \vdash B : s_j}{\Gamma \vdash \Pi x^A .\ B : s_j}$$

if $i \in \mathcal{I}$, then $\tau(\Gamma) = \tau(\Gamma, x : A)$ and $\tau_\Gamma(\Pi x^A.\ B) = \tau_{\Gamma,x:A}(B)$ and so $\tau(\Gamma) \vdash \tau_{\Gamma,x:A}(B) : s_j$ by the inductive hypothesis applied to the right antecedent judgment. It cannot be the case that $i - 1 \in \mathcal{I}$ and $A = s_{i-1}$ since s_i is rule-isolated in this case. The remaining case is straightforward.

Abstraction. Suppose the last inference is of the form

$$\frac{\Gamma, {}^{s_i}x : A \vdash M : B \qquad \Gamma \vdash \Pi x^A.\ B : s_j}{\Gamma \vdash \lambda x^A.\ M : \Pi x^A.\ B}$$

If $i \in \mathcal{I}$, then

$$\begin{aligned} \tau(\Gamma) &= \tau(\Gamma, {}^{s_i}x : A) \\ \tau_\Gamma(\lambda x^A.\ M) &= \tau_{\Gamma,x:A}(M) \\ \tau_\Gamma(\Pi x^A.\ B) &= \tau_{\Gamma,x:A}(B) \end{aligned}$$

so the desired judgment follows directly from the inductive hypothesis applied to the left antecedent judgment. Again, it cannot be the case that $i - 1 \in \mathcal{I}$ and $A = s_{i-1}$ since s_i is rule-isolated in this case. The remaining case is straightforward.

Application. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : \Pi x^A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/^{s_i}x]}$$

By type correctness, $\Gamma \vdash \Pi x^A.\ B : s_j$ for some sort s_j, and by generation, we have

$$\Gamma, {}^{s_i}x : A \vdash B : s_j$$

so by Lemma 28, if $i \in \mathcal{I}$ (i.e., $\deg(N) + 1 \in \mathcal{I}$), then $\tau_\Gamma(MN) = \tau_\Gamma(M)$ and

$$\tau_\Gamma(B[N/^{s_i}x]) = \tau_{\Gamma,x:A}(B) = \tau_\Gamma(\Pi x^A.\ B).$$

The desired result then follows directly from the inductive hypothesis applied to the left antecedent judgment. And if $i \notin \mathcal{I}$, then $\tau_\Gamma(B[N/x]) = \tau_{\Gamma,x:A}(B)[\tau_\Gamma(N)/x]$ and we have

$$\frac{\tau(\Gamma) \vdash \tau_\Gamma(M) : \Pi x^{\tau_\Gamma(A)}.\ \tau_{\Gamma,x:A}(B) \qquad \tau(\Gamma) \vdash \tau_\Gamma(N) : \tau_\Gamma(A)}{\tau(\Gamma) \vdash \tau_\Gamma(M)\tau_\Gamma(N) : \tau_{\Gamma,x:A}(B)[\tau_\Gamma(N)/x]}$$

Conversion. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s_i}{\Gamma \vdash M : B}$$

where $A =_\beta B$. Then we have

$$\frac{\tau(\Gamma) \vdash \tau_\Gamma(M) : \tau_\Gamma(A) \qquad \tau(\Gamma) \vdash \tau_\Gamma(B) : s_i}{\tau(\Gamma) \vdash \tau_\Gamma(M) : \tau_\Gamma(B)}$$

where $\tau_\Gamma(A) =_\beta \tau_\Gamma(B)$ by Lemma 30. ◀

It remains to show that this translation is path-preserving. The guiding observation is that β_2-reductions cannot make more "complex" redexes. We define a complexity measure which captures this observation by its being monotonically decreasing in β_2-reductions.

▶ **Definition 33.** *The* ***shallow*** λ***-depth*** *of an expression M is the number of top-level λ's appearing in it, i.e., the function $\delta : \mathsf{T} \to \mathbb{N}$ is given by $\delta(\lambda x^A.\ N) \triangleq 1 + \delta(N)$ and $\delta(M) \triangleq 0$ otherwise. The* ***shallow*** λ***-depth of a redex*** *$(\lambda x^A.\ M)N$ is the shallow λ-depth of its left term $\lambda x^A.\ M$. I will simply write "depth" from this point forward.*

▶ **Definition 34.** *Define $\mu : \mathsf{T} \to \mathbb{N}$ to be the function which maps an expression to the sum of the depths of its β_2-redexes, i.e.,*

$$\mu(s_i) = \mu(x) \triangleq 0$$
$$\mu(\Pi x^A.\ B) = \mu(\lambda x^A.\ B) \triangleq \mu(A) + \mu(B)$$
$$\mu(MN) \triangleq \begin{cases} \mu(M) + \mu(N) + \delta(MN) & MN \text{ is a } \beta_2\text{-redex} \\ \mu(M) + \mu(N) & \textit{otherwise.} \end{cases}$$

Finally, we prove the monotonicity lemma. It depends on the domain-full version of a result by Lévy for the untyped lambda calculus about the creation of new redexes [14]. I give the statement of the result here without proof (See [11, 20] among others for the standard definition of a residual).

▶ **Lemma 35.** *For expressions M and N such that $M \to_\beta N$, if $(\lambda x^A.\ P)Q$ is a redex of N which is not a residual of a redex in M, then it is created in one of the following ways.*

1. $(\lambda y^B.\ y)(\lambda x^A.\ P)Q \to_\beta (\lambda x^A.\ P)Q$;
2. $(\lambda y^C.\ \lambda x^D.\ R)SQ \to_\beta (\lambda x^{D[S/y]}.\ R[S/y])Q$ *where* $A = D[S/y]$ *and* $P = R[S/y]$;
3. $(\lambda y^B.\ R)(\lambda x^A.\ P) \to_\beta R[\lambda x^A.\ P/y]$ *where yQ is a sub-expression of R.*

▶ **Lemma 36.** *For derivable expressions M and N, if $M \to_{\beta_2} N$, then $\mu(M) > \mu(N)$.*

Proof. Suppose M reduces to N by reducing the β_2-redex $(\lambda x^C.\ P)Q$. By Corollary 24, the expression Q is of the form $\Pi {x_1}^{A_1}.\ \ldots \Pi {x_k}^{A_k}.\ B$ where $\deg(A_j) \in \mathcal{I}$ for all j and either $B = s_{i-2}$ or $B \in \mathsf{V}_{s_i}$. This means reducing a β_2-redex cannot duplicate existing redexes in M, so every redex has at most one residual in N. Furthermore, if N has a new β_2-redex, it is by item 2 of Lemma 35, i.e., there are expressions C, D, R, and S, and variable z such that $P = \lambda z^D.\ R$ and

$$(\lambda x^C.\ \lambda z^D.\ R)QS \to_\beta (\lambda z^{D[Q/x]}.\ R[Q/x])S.$$

It is easy to verify that, because of the form of Q, only one new β-redex is created and, furthermore, $\delta(R[Q/x]) \le \delta(R)$. This implies the new redex has smaller depth than the redex that was reduced, so even if it is a β_2-redex, the complexity of M decreases. ◀

The proof of the main theorem of this section is standard.

▶ **Theorem 37.** *If $\lambda\mathcal{S}^-$ is strongly normalizing, then $\lambda\mathcal{S}$ is strongly normalizing.*

Proof. Suppose there is an infinite reduction sequence in $\lambda\mathcal{S}$

$$M_1 \to_\beta M_2 \to_\beta \ldots$$

where M_1 is derivable in $\lambda\mathcal{S}$ from the context Γ. Since μ is monotonically decreasing in β_2-reductions (Lemma 36), there cannot be an infinite sequence of solely β_2-reductions contained in this sequence. This means there are infinitely many β_1 reductions in this sequence, which by Lemma 30 implies there infinitely many β-reductions in the reduction path

$$\tau_\Gamma(M_1) \twoheadrightarrow_\beta \tau_\Gamma(M_2) \twoheadrightarrow_\beta \ldots$$

where $\tau_\Gamma(M_1)$ is derivable in $\lambda\mathcal{S}^-$ by Lemma 32. ◀

3.2 Eliminating Completely Isolated Sorts

We now handle completely isolated sorts. Recall that a sort s_i is completely isolated if s_i is top-sort-like and rule-isolated. This translation is slightly simpler than the first. It is a generalization of the observation made in the introduction that one can define a path-preserving translation from $\lambda\mathsf{HOL}$ to $\lambda\omega$, i.e., one that eliminates the rule-isolated top-sort.

Fix an n-tiered pure type system $\lambda\mathcal{S}$ with $n > 2$, and a completely isolated sort s_i.[7] In essence, the following translation removes the completely isolated sort and shifts down all the sorts that might be above it. Because isolated sorts can only really be used to introduce variables into the context, the translation pre-substitutes those variables with dummy values that won't affect the normalization behavior of the expression after translation.

[7] The restriction on n is a technicality that ensures the target system is nontrivial. See, for example, the variable case of Definition 38.

One notable feature of this translation is that it does not preserve the number of sorts in the system and, furthermore, does not preserve degree. It will be useful to be more careful about variable annotations in the following definitions and lemmas.

▶ **Definition 38.** *Define the context-indexed function $\theta_\Gamma : \mathsf{Ctx} \times \mathsf{T} \to \mathsf{T}$ inductively on both arguments as follows.*

$$\theta_\Gamma(s_j) \triangleq \begin{cases} s_j & j < i \\ s_{j-1} & \textit{otherwise} \end{cases}$$

$$\theta_\Gamma({}^{s_j}x) \triangleq \begin{cases} s_{i-2} & \textit{if } j = i \textit{ and } ({}^{s_i}x : s_{i-1}) \in \Gamma \\ {}^{s_j}x & j < i \\ {}^{s_{j-1}}x & \textit{otherwise} \end{cases}$$

$$\theta_\Gamma(\Pi^{s_j}x^A.\ B) \triangleq \Pi^{\theta_\Gamma(s_j)}x^{\theta_\Gamma(A)}.\ \theta_{\Gamma,x:A}(B)$$

$$\theta_\Gamma(\lambda^{s_j}x^A.\ M) \triangleq \lambda^{\theta_\Gamma(s_j)}x^{\theta_\Gamma(A)}.\ \theta_{\Gamma,x:A}(M)$$

$$\theta_\Gamma(MN) \triangleq \theta_\Gamma(M)\theta_\Gamma(N)$$

This function is used to define a function on contexts as

$$\theta(\varnothing) \triangleq \varnothing$$

$$\theta(\Gamma, {}^{s_j}x : A) \triangleq \begin{cases} \theta(\Gamma) & \textit{if } j = i \textit{ and } A = s_{i-1} \\ \theta(\Gamma), {}^{\theta_\Gamma(s_j)}x : \theta_\Gamma(A) & \textit{otherwise.} \end{cases}$$

As with the previous translation, contexts can be weakened without changing the value of the function (in analogy with Proposition 27 for τ_Γ). We go on to prove substitution-commutation, β-reduction preservation, and typability preservation. The proofs are similar to those in the previous sub-section and, consequently, are slightly abbreviated.

▶ **Lemma 39.** *For context Γ, expressions M, N, A and B, and variable ${}^{s_j}x$, if $j \neq i$ and $\Gamma, {}^{s_j}x : A \vdash M : B$ and $\Gamma \vdash N : A$ then $\theta_\Gamma(M[N/{}^{s_j}x]) = \theta_{\Gamma,{}^{s_j}x:A}(M)[\theta_\Gamma(N)/{}^{\theta_\Gamma(s_j)}x]$.*

Proof. By induction on the structure of M. All cases are straightforward except the case in which M is a variable, but then the assumption that $j \neq i$ ensures the desired equality holds. ◀

▶ **Lemma 40.** *For expressions M and N derivable from Γ, if $M \to_\beta N$, then $\theta_\Gamma(M) \to_\beta \theta_\Gamma(N)$. Furthermore, if $M =_\beta N$, then $\theta_\Gamma(M) =_\beta \theta_\Gamma(N)$.*

Proof. The second part follows directly from the first, which follows by induction on the structure of the one-step β-reduction relation. In the case of a redex $(\lambda x^A.\ M)N$, we have

$$\begin{aligned} \theta_\Gamma((\lambda x^A.\ M)N) &= (\lambda x^{\theta_\Gamma(A)}.\ \theta_{\Gamma,x:A}(M))\theta_\Gamma(N) \\ &\to_\beta \theta_{\Gamma,x:A}(M)[\theta_\Gamma(N)/{}^{\theta_\Gamma(s_j)}x] \\ &= \theta_\Gamma(M[N/{}^{\theta_\Gamma(s_j)}x]) \end{aligned}$$

where the last equality follows from Lemma 39, keeping in mind that $j \neq i$ since i is isolated, so the lemma can be safely applied. ◀

Finally, typability preservation. The target system is as expected, the completely isolated sort s_i is removed and potential sorts above it are shifted down.

▶ **Definition 41.** *The i-**collapse** of $\lambda\mathcal{S}$, denote here by $\lambda\mathcal{S}^*$, is the $(n-1)$-tiered systems specified by the rules $\{(\theta_\varnothing(s_j), \theta_\varnothing(s_k)) \mid (s_j, s_k) \in \mathcal{R}_{\lambda\mathcal{S}}\}$.*

▶ **Lemma 42.** *For context Γ and expressions M and A where $M \neq s_{i-1}$, if*

$$\Gamma \vdash M : A \qquad \textit{then} \qquad \theta(\Gamma) \vdash \theta_\Gamma(M) : \theta_\Gamma(A).$$

Proof. By induction on the structure of derivations. The proof differs slightly depending on whether or not s_i is a top-sort. I make clear below which cases differ.

Axiom. Since $M \neq s_{i-1}$, the judgment $\varnothing \vdash \theta_\varnothing(s_j) : \theta_\varnothing(s_{j+1})$ is still an axiom.

Variable Introduction. Suppose the last inference is of the form

$$\frac{\Gamma \vdash A : s_j}{\Gamma, {}^{s_j}x : A \vdash {}^{s_j}x : A}$$

If $j = i$ and $A = s_{i-1}$, then $\theta(\Gamma) \vdash s_{i-2} : s_{i-1}$ is still derivable. Note that $\theta(\Gamma)$ can be proved to be well-formed by the inductive hypothesis. If $j < i$, then we have

$$\frac{\theta(\Gamma) \vdash \theta_\Gamma(A) : s_j}{\theta(\Gamma), {}^{s_j}x : \theta_\Gamma(A) \vdash {}^{s_j}x : \theta_\Gamma(A)}$$

If $j > i$, then in particular s_i is not a top-sort. This case is then similar to the previous one, keeping in mind that this might use the axiom (s_{i-1}, s_i) for the translated derivation *in the system* $\lambda\mathcal{S}^*$, but not in the case that s_i is a top-sort.

Weakening. This case follows directly from the fact that $\theta_{\Gamma,x:B}(M) = \theta_\Gamma(M)$ whenever M and B are derivable from Γ. It is also similar to the analogous case in the previous sub-section.

Product Type Formation. Suppose the last inference is of the form

$$\frac{\Gamma \vdash A : s_j \qquad \Gamma, {}^{s_j}x : A \vdash B : s_k}{\Gamma \vdash \Pi x^A.\ B : s_k}$$

Note that $j \neq i$ and $k \neq i$ since s_i is rule-isolated. In particular, neither A nor B are s_{i-1}. Therefore, we can apply the inductive hypothesis directly to each antecedent judgment and derive the desired consequent judgment.

Abstraction. Suppose the last inference is of the form

$$\frac{\Gamma, {}^{s_j}x : A \vdash M : B \qquad \Gamma \vdash \Pi x^A.\ B : s_k}{\Gamma \vdash \lambda x^A.\ M : \Pi x^A.\ B}$$

Note that $j \neq i$ since s_i is rule-isolated, and so $\Pi x^A.\ B$ would not be derivable. Furthermore, $B \neq s_i$ (so $M \neq s_{i-1}$) since s_i is irrelevant. Therefore, we can apply the inductive hypothesis directly to each antecedent judgment and derive the desired consequent judgment.

Application. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : \Pi x^A.\ B \qquad \Gamma \vdash N : A}{\Gamma \vdash MN : B[N/x]}$$

Note that $\deg(A) \neq i+1$ (and in particular $N \neq s_{i-1}$), since s_{i+1} is rule-isolated. Furthermore, $\deg(A) \neq i$ (and $\deg(N) \neq i-1$) since s_i is rule-isolated. Therefore, we can apply the inductive hypothesis directly to each antecedent judgment to derive

$$\theta(\Gamma) \vdash \theta_\Gamma(M)\theta_\Gamma(N) : \theta_{\Gamma,x:A}(B)[\theta_\Gamma(N)/x]$$

where $\theta_{\Gamma,x:A}(B)[\theta_\Gamma(N)/x] = \theta_\Gamma(B[N/x])$ by Lemma 39.

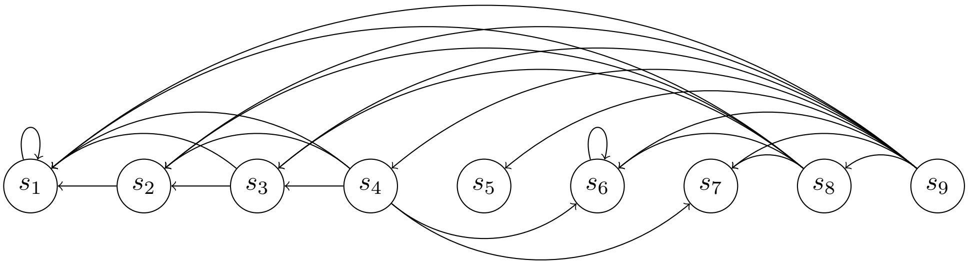


Figure 2 A system with a non-trivial sequence of irrelevance reductions.

Conversion. Suppose the last inference is of the form

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : s_j}{\Gamma \vdash M : B}$$

If $M = s_{i-1}$, then $A =_\beta s_i =_\beta B$. Then by Lemma 21, in fact $A = B$. If $B = s_{i-1}$, then by Corollary 24 we again have $A = B$. Otherwise, by Lemma 40, $\theta_\Gamma(A) =_\beta \theta_\Gamma(B)$ and we can derive $\theta(\Gamma) \vdash \theta_\Gamma(M) : \theta_\Gamma(B)$ by the inductive hypothesis and conversion. ◀

Since β-reductions are simulated directly, the argument for the final theorem is straightforward.

▶ **Theorem 43.** *If $\lambda\mathcal{S}^*$ is strongly normalizing then $\lambda\mathcal{S}$ is strongly normalizing.*

3.3 The Final Translation

We can now consider the fixed-points of the above translations.

▶ **Definition 44.** *Let $\tau(\lambda\mathcal{S})$ denote the fixed-point of taking the irrelevance reduction of $\lambda\mathcal{S}$ with respect to maximum completely irrelevant index sets. That is, repeat $\lambda\mathcal{S} := \lambda\mathcal{S}^-$ taken with respect to the maximum completely irrelevant index set of $\lambda\mathcal{S}$, until its maximum completely irrelevant index set is empty.*

▶ **Definition 45.** *Let $\theta(\lambda\mathcal{S})$ denote the fixed-point of taking the i-collapse of $\lambda\mathcal{S}$, where i is the maximum index of a complete isolated sort in $\lambda\mathcal{S}$, if one exists. That is, repeat $\lambda\mathcal{S} := \lambda\mathcal{S}^*$ taken with respect to the maximum index of a completely isolated sort of $\lambda\mathcal{S}$ until it has no completely isolated sort or is 2-tiered.*

Note that a sort which does not appear in the maximum completely irrelevant index set of $\lambda\mathcal{S}$ may appear in the maximum completely irrelevant set of $\lambda\mathcal{S}^-$. See Figure 2 for a tiered system with a non-trivial sequence of irrelevance reductions. The maximum completely irrelevant index set of this system is $\{9\}$, but after eliminating the rules associated with s_9, both s_9 and s_5 become rule-isolated, and so the next maximum completely irrelevant index set is $\{4, 8\}$. One can then imagine how this effect can be scaled up to larger systems.

I will write $\lambda\mathcal{S}^\downarrow$ for $\tau(\lambda\mathcal{S})$ and $\lambda\mathcal{S}^\Downarrow$ for $\theta(\tau(\lambda\mathcal{S}))$. Since no rules are removed by an i-collapse (only shifted), no sort can become completely isolated and no new completely irrelevant index set can be created, so in fact $\lambda\mathcal{S}^\Downarrow$ is the fixed-point of $\theta \circ \tau$.

The main two theorems are as follows.

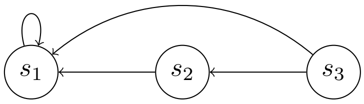
▶ **Theorem 46.** *For any tiered pure type system $\lambda\mathcal{S}$, if $\lambda\mathcal{S}^\Downarrow$ is strongly normalizing, then $\lambda\mathcal{S}$ is strongly normalizing.*

▶ **Theorem 47.** *For any tiered pure type system $\lambda\mathcal{S}$, if weak normalization implies strong normalization for $\lambda\mathcal{S}^{\downarrow}$, then weak normalization implies strong normalization for $\lambda\mathcal{S}$.*

In particular, if $\lambda\mathcal{S}^{\downarrow}$ satisfies the conditions of Barthe et al. ([4], Theorem 5.21), then weak normalization implies strong normalization in $\lambda\mathcal{S}$. This does not immediately apply to $\lambda\mathcal{S}^{\Downarrow}$ since it is not immediate that weak normalization is preserved from $\lambda\mathcal{S}$ to $\lambda\mathcal{S}^{*}$; the sorts are not preserved. Note that it is immediate in the case that the completely isolated sort is a top-sort. Given the scope of this work, I leave this to be verified, it is a natural step in extending these results.

4 Conclusions

I have presented a path-preserving translation which eliminates some irrelevant structure. Again, this structure is irrelevant with respect to normalization, not derivability. When combined with results for the Barendregt-Geuvers-Klop conjecture, it widens the class of systems for which the conjecture applies. This is a step towards proving the conjecture for all tiered systems, in particular because it highlights those systems which require further analysis. For example, it appears that dealing with circular rules is one of the clear barriers in strengthening these results. For 3-tiered systems, we extend the conjecture to (and can prove strong normalization of) the system[8]



but not to the same system with the additional rule (s_3, s_3). Circular rules break irrelevancy and, consequently, induce much more complicated structure in the system.

Additionally, it is worth noting that the conditions on completely irrelevant index sets cannot be trivially weakened. If, for example the irrelevance condition on preceding sorts was removed, this technique would apply to λU (i.e., the same system presented above but with the additional rule (s_2, s_2)), leading to a contradiction since λU is non-normalizing. Circular rules again seem to be at the core of this issue. More carefully considering λU and related non-normalizing systems through the lens of these results – particularly why the techniques don't apply to these systems – may yield a more structural understanding of the non-normalization of λU. Regardless, I hope to have demonstrated with this translation that, despite the full Barendregt-Geuvers-Klop conjecture seeming quite far from being resolved, there are still a number of approachable questions and avenues for further development.

References

1 TLCA List of Open Problems, 2014. `http://tlca.di.unito.it/opltlca/`.

2 Henk Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

3 Henk Barendregt. Lambda Calculi with Types. In *Handbook of Logic in Computer Science, Volume II*, pages 117–309. Oxford University Press, 1993.

4 Gilles Barthe, John Hatcliff, and Morten Heine Sørensen. Weak normalization implies strong normalization in a class of non-dependent pure type systems. *Theoretical Computer Science*, 269(1-2):317–361, 2001.

[8] This system is not covered by the Barthe et al. result because s_2 is not negatable.

5 Stefano Berardi. Towards a mathematical analysis of the Coquand-Huet calculus of constructions and the other systems in Barendregt's cube. Technical report, Carnegie Mellon University, Universita di Torino, 1988.

6 Stefano Berardi. *Type Dependence and Constructive Mathematics*. PhD thesis, Dipartimento di Informatica, Torino, Italy, 1990.

7 Herman Geuvers. The Calculus of Constructions and Higher Order Logic. In *The Curry-Howard Isomorphism*, volume 8 of *Cahiers du Centre de Logique (Universit'e catholique de Louvain), Academia, Louvain-la-Neuve (Belgium)*, pages 131–191, 1995.

8 Herman Geuvers and Mark-Jan Nederhof. Modular proof of strong normalization for the calculus of constructions. *Journal of Functional Programming*, 1(2):155–189, 1991.

9 Jan Herman Geuvers. *Logic and Type Systems*. PhD thesis, University of Nijmegen, 1993.

10 Robert Harper, Furio Honsell, and Gordon Plotkin. A Framework for Defining Logics. *Journal of the ACM*, 40(1):143–184, 1993.

11 Gérard Huet. Residual Theory in λ-Calculus: A Formal Development. *Journal of Functional Programming*, 4(3):371–394, 1994.

12 Fairouz Kamareddine, Twan Laan, and Rob Nederpelt. *A Modern Perspective on Type Theory: From its Origins until Today*, volume 29 of *Applied Logic Series*. Springer, 2004.

13 Jeroen Ketema, Jan Willem Klop, and V van Oostrom. Vicious Circles in Rewriting Systems. *Artificial Intelligence Preprint Series*, 52, 2004.

14 Jean-Jacques Lévy. *Réductions Correctes et Optimales dans le Lambda-Calcul*. PhD thesis, L'Université Paris VII, 1978.

15 Zhaohui Luo. *An extended calculus of constructions*. PhD thesis, University of Edinburgh, 1990.

16 Cody Roux and Floris van Doorn. The Structural Theory of Pure Type Systems. In *Rewriting and Typed Lambda Calculi*, pages 364–378. Springer, 2014.

17 Morten Heine Sørensen. Strong Normalization from Weak Normalization in Typedλ-Calculi. *Information and Computation*, 133(1):35–71, 1997.

18 Jan Terlouw. Een nadere bewijstheoretische analyse van GSTT's. Technical report, Department of Computer Science, University of Nijmege, 1989.

19 Hongwei Xi. On Weak and Strong Normalisations. Technical report, Carnegie Mellon University, Department of Mathematics, 1996.

20 Hongwei Xi. Development Separation in Lambda-Calculus. *Electronic Notes in Theoretical Computer Science*, 143:207–221, 2006.

Linear Rank Intersection Types

Fábio Reis
DCC-FCUP, University of Porto, Portugal
LIACC – Artificial Intelligence and Computer Science Laboratory, University of Porto, Portugal

Sandra Alves
DCC-FCUP, University of Porto, Portugal
LIACC – Artificial Intelligence and Computer Science Laboratory, University of Porto, Portugal
CRACS, INESC-TEC – Centre for Research in Advanced Computing Systems, Porto, Portugal

Mário Florido
DCC-FCUP, University of Porto, Portugal
LIACC – Artificial Intelligence and Computer Science Laboratory, University of Porto, Portugal

Abstract

Non-idempotent intersection types provide quantitative information about typed programs, and have been used to obtain time and space complexity measures. Intersection type systems characterize termination, so restrictions need to be made in order to make typability decidable. One such restriction consists in using a notion of finite rank for the idempotent intersection types. In this work, we define a new notion of rank for the non-idempotent intersection types. We then define a novel type system and a type inference algorithm for the λ-calculus, using the new notion of rank 2. In the second part of this work, we extend the type system and the type inference algorithm to use the quantitative properties of the non-idempotent intersection types to infer quantitative information related to resource usage.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Lambda-Calculus, Intersection Types, Quantitative Types, Tight Typings

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.8

Related Version *Full Version*: `https://arxiv.org/abs/2211.17186`

Funding *Fábio Reis*: This work was partially financially supported by Base Funding – UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC – funded by national funds through the FCT/MCTES (PIDDAC).
Sandra Alves: Partially supported by National Funds through the Portuguese funding agency, FCT – Fundação para a Ciência e a Tecnologia, within project LA/P/0063/2020.
Mário Florido: This work was partially financially supported by Base Funding – UIDB/00027/2020 of the Artificial Intelligence and Computer Science Laboratory – LIACC – funded by national funds through the FCT/MCTES (PIDDAC).

1 Introduction

The ability to determine upper bounds for the number of execution steps of a program in compilation time is a relevant problem, since it allows us to know in advance the computational resources needed to run the program.

Type systems are a powerful and successful tool of static program analysis that are used, for example, to detect errors in programs before running them. Quantitative type systems, besides helping on the detection of errors, can also provide quantitative information related to computational properties.

Intersection types, defined by the grammar $\sigma ::= \alpha \mid \sigma_1 \cap \cdots \cap \sigma_n \to \sigma$ (where α is a type variable and $n \geq 1$), are used in several type systems for the λ-calculus [6, 7, 18, 27] and allow λ-terms to have more than one type. Non-idempotent intersection types [16, 20, 12, 4], also known as *quantitative types*, are a flavour of intersection types in which the type constructor

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 8; pp. 8:1–8:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$\cap$ is non-idempotent, and provide more than just qualitative information about programs. They are particularly useful in contexts where we are interested in measuring the use of resources, as they are related to the consumption of time and space in programs. Type systems based on non-idempotent intersection types, use non-idempotence to count the number of evaluation steps and the size of the result. For instance in [1], the authors define several quantitative type systems, corresponding to different evaluation strategies, for which they are able to measure the number of steps taken by that strategy to reduce a term to its normal form, and the size of the term's normal form. Typability is undecidable for intersection type systems, as it corresponds to termination. One way to get around this is to restrict intersection types to finite ranks, a notion defined by Daniel Leivant in [21] that makes typability decidable - Kfoury and Wells [19] define an intersection type system that, when restricted to any finite-rank, has principal typings and decidable type inference. Type systems that use finite-rank intersection types are still very powerful and useful. For instance, rank 2 intersection type systems [18, 26, 11] are more powerful, in the sense that they can type strictly more terms, than popular systems like the ML type system [10]. Still related to decidability of typability for finite ranks, Dudenhefner and Rehof [14] studied the problem for a notion of bounded-dimensional intersection types. This notion was previously defined in the context of type inhabitation [13], where it was used to prove decidability of type inhabitation for a non-idempotent intersection type system (the problem is known to be undecidable above rank 2, for idempotent intersection types [25]).

In this paper we present a new definition of rank for the quantitative types, which we call *linear rank* and differs from the classical one in the base case – instead of simple types, linear rank 0 intersection types are the linear types. In a non-idempotent intersection type system, every linear term is typable with a simple type (in fact, in many of those systems, only the linear terms are), which is the motivation to use linear types for the base case. The relation between non-idempotent intersection types and linearity has already been studied by Kfoury [20], de Carvalho [12], Gardner [16] and Florido and Damas [15]. Our motivation to redefine rank in the first place, has to do with our interest in using non-idempotent intersection types to estimate the number of evaluation steps of a λ-term to normal form while inferring its type, and the realization that there is a way to define rank that is more suitable for the quantitative types. We define a new intersection type system for the λ-calculus, restricted to linear rank 2 non-idempotent intersection types, and a new type inference algorithm that we prove to be sound and complete with respect to the type system.

Finally we extend our type system and inference algorithm to use the quantitative properties of the linear rank 2 non-idempotent intersection types to infer not only the type of a λ-term, but also the number of evaluation steps of the term to its normal form. The new type system is the result of a merge between our Linear Rank 2 Intersection Type System and the system for the leftmost-outermost evaluation strategy presented in [1]. The type system in [1] is a quantitative typing system extended with the notion of tight types (which provide an effective characterisation of minimal typings) that is crucial to extract exact bounds for reduction. We prove that the system gives the correct number of evaluation steps for a kind of derivation. As for the new type inference algorithm, we show that it is sound and complete with respect to the type system for the inferred types, and conjecture that the inferred measures correspond to the ones given by the type system (i.e., correspond to the number of evaluation steps of the term to its normal form, when using the leftmost-outermost evaluation strategy).

Thus, the main contributions of this paper are the following:

- A new definition of rank for non-idempotent intersection types, which we call *linear rank* (Section 3);
- A Linear Rank 2 Intersection Type System for the λ-calculus (Section 3);

- A type inference algorithm that is sound and complete with respect to the Linear Rank 2 Intersection Type System (Section 3);
- A Linear Rank 2 Quantitative Type System for the λ-calculus that derives a measure related to the number of evaluation steps for the leftmost-outermost strategy (Section 4);
- A type inference algorithm that is sound and complete with respect to the Linear Rank 2 Quantitative Type System, for the inferred types, and gives a measure that we conjecture to correspond to the number of evaluation steps of the typed term for the leftmost-outermost strategy (Section 4).

In this paper we assume that the reader is familiar with the λ-calculus [3]. From now on, in the rest of the paper, terms of the λ-calculus are considered modulo α-equivalence and we use Barendregt's variable convention [2].

2 Intersection Types

The simply typed λ-calculus is a typed version of the λ-calculus, introduced by Alonzo Church in [5] and by Haskell Curry and Robert Feys in [9]. One system that uses simple types is the Curry Type System, which was first introduced in [8] for the theory of combinators, and then modified for the λ-calculus in [9]. Typability in this system is decidable and there is an algorithm that given a term, returns its principal pair. However, the system presents some disadvantages when comparing to others, one of them being the large number of terms that cannot be typed. For example, in the Curry Type System we cannot assign a type to the λ-term $\lambda x.xx$. This term, on the other hand, can be typed in systems that use intersection types, which allow terms to have more than one type. Such a system is the Coppo-Dezani Type System [6], which was one of the first to use intersection types, and a basis for subsequent systems.

▶ **Definition 1** (Intersection types). *Intersection types $\sigma, \sigma_1, \sigma_2, \ldots \in \mathbb{T}$ are defined by the following grammar:*

$$\sigma ::= \alpha \mid \sigma_1 \cap \cdots \cap \sigma_n \to \sigma$$

where $n \geq 1$ and $\sigma_1 \cap \cdots \cap \sigma_n$ is called a sequence *of types.*

Note that intersections arise in different systems in different scopes. Here we follow several previous presentations where intersections are only allowed directly on the left-hand side of arrow types and sequences are non-empty [6, 7, 18, 27].

▶ **Notation.** *The intersection type constructor $\cap$ binds stronger than $\to$: $\alpha_1 \cap \alpha_2 \to \alpha_3$ stands for $(\alpha_1 \cap \alpha_2) \to \alpha_3$.*

▶ **Example 2.** Some examples of intersection types are:

α;
$\alpha_1 \to \alpha_2$;
$\alpha_1 \cap \alpha_2 \to \alpha_3$;
$(\alpha_1 \cap \alpha_2 \to \alpha_3) \to \alpha_4$;
$\alpha_1 \cap (\alpha_1 \to \alpha_2) \to \alpha_3$.

▶ **Definition 3** (Coppo-Dezani Type System). *In the Coppo-Dezani Type System, we say that M has type σ given the environment Γ (where the predicates of declarations are sequences), and write $\Gamma \vdash_{\mathcal{CD}} M : \sigma$, if $\Gamma \vdash_{\mathcal{CD}} M : \sigma$ can be obtained from the* derivation rules *in Figure 1, where $1 \leq i \leq n$:*

$$\Gamma \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} x : \sigma_i \qquad \text{(Axiom)}$$

$$\frac{\Gamma \cup \{x : \sigma_1 \cap \cdots \cap \sigma_n\} \vdash_{\mathcal{CD}} M : \sigma}{\Gamma \vdash_{\mathcal{CD}} \lambda x.M : \sigma_1 \cap \cdots \cap \sigma_n \to \sigma} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash_{\mathcal{CD}} M_1 : \sigma_1 \cap \cdots \cap \sigma_n \to \sigma \qquad \Gamma \vdash_{\mathcal{CD}} M_2 : \sigma_1 \ \cdots \ \Gamma \vdash_{\mathcal{CD}} M_2 : \sigma_n}{\Gamma \vdash_{\mathcal{CD}} M_1 M_2 : \sigma} \qquad (\to \text{Elim})$$

Figure 1 Coppo-Dezani Type System.

▶ **Example 4.** For the λ-term $\lambda x.xx$ the following derivation is obtained:

$$\frac{\dfrac{\{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} x : \sigma_1 \to \sigma_2 \qquad \{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} x : \sigma_1}{\{x : \sigma_1 \cap (\sigma_1 \to \sigma_2)\} \vdash_{\mathcal{CD}} xx : \sigma_2}}{\vdash_{\mathcal{CD}} \lambda x.xx : \sigma_1 \cap (\sigma_1 \to \sigma_2) \to \sigma_2}$$

This system is a true extension of the Curry Type System, allowing term variables to have more than one type in the (→ Intro) derivation rule and the right-hand term to also have more than one type in the (→ Elim) derivation rule.

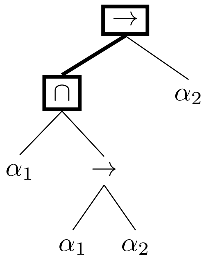
2.1 Finite Rank

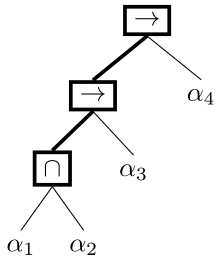
Intersection type systems, like the Coppo-Dezani Type System, characterize termination, in the sense that a λ-term is strongly normalizable if and only if it is typable in an intersection type system. Thus, typability is undecidable for these systems.

To get around this, some current intersection type systems are restricted to types of finite rank [18, 26, 19, 11] using a notion of rank first defined by Daniel Leivant in [21]. This restriction makes typability decidable [19]. Despite using finite-rank intersection types, these systems are still very powerful and useful. For instance, rank 2 intersection type systems [18, 26, 11] are more powerful, in the sense that they can type strictly more terms, than popular systems like the ML type system [10].

The *rank* of an intersection type is related to the depth of the nested intersections and it can be easily determined by examining the type in tree form: a type is of rank k if no path from the root of the type to an intersection type constructor ∩ passes to the left of k arrows.

▶ **Example 5.** The intersection type $\alpha_1 \cap (\alpha_1 \to \alpha_2) \to \alpha_2$ (tree on the left) is a rank 2 type and $(\alpha_1 \cap \alpha_2 \to \alpha_3) \to \alpha_4$ (tree on the right) is a rank 3 type:





▶ **Definition 6** (Rank of intersection types). *Let $\mathbb{T}_0$ be the set of simple types and $\mathbb{T}_1 = \{\tau_1 \cap \cdots \cap \tau_m \mid \tau_1, \ldots, \tau_m \in \mathbb{T}_0, m \geq 1\}$ the set of sequences of simple types (written as $\vec{\tau}$). The set $\mathbb{T}_k$, of rank k intersection types (for $k \geq 2$), can be defined recursively in the following way ($n \geq 3$, $m \geq 1$):*

$$\mathbb{T}_2 = \mathbb{T}_0 \cup \{\vec{\tau} \to \sigma \mid \vec{\tau} \in \mathbb{T}_1, \sigma \in \mathbb{T}_2\}$$
$$\mathbb{T}_n = \mathbb{T}_{n-1} \cup \{\rho_1 \cap \cdots \cap \rho_m \to \sigma \mid \rho_1, \ldots, \rho_m \in \mathbb{T}_{n-1}, \sigma \in \mathbb{T}_n\}$$

▶ **Notation.** *We consider the intersection type constructor $\cap$ to be associative, commutative and non-idempotent (meaning that $\alpha \cap \alpha$ is not equivalent to α).*

We are particularly interested in non-idempotent intersection types, also known as quantitative types, because they provide more quantitative information than the idempotent ones.

3 Linear Rank Intersection Types

In the previous chapter, we mentioned several intersection type systems in which intersection is idempotent and types are rank-restricted. There are also many quantitative type systems [16, 20, 12, 4] that, on the other hand, make use of non-idempotent intersection types, for which there is no specific definition of rank.

The generalization of ranking for non-idempotent intersection types is not trivial and raises interesting questions that we will address in this chapter, along with a definition of a new non-idempotent intersection type system and a type inference algorithm.

This and the following sections cover original work that we presented at the TYPES 2022 conference [23].

3.1 Linear Rank

The set of terms typed using idempotent rank 2 intersection types and non-idempotent rank 2 intersection types is not the same. For instance, the term $(\lambda x.xx)(\lambda fx.f(fx))$ is typable with a simple type when using idempotent intersection types, but not when using non-idempotent intersection types. This comes from the two different occurrences of f in $\lambda fx.f(fx)$, which even if typed with the same type, are not contractible because intersection is non-idempotent. Note that this is strongly related to the linearity features of terms. A λ-term M is called a *linear term* if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N exactly once, and if each free variable of M has just one occurrence free in M. So the term $(\lambda x.xx)(\lambda fx.f(fx))$ is not typable with a non-idempotent rank 2 intersection type precisely because the term $\lambda fx.f(fx)$ is not linear.

Note that in a non-idempotent intersection type system, every linear term is typable with a simple type (in fact, in many of those systems, only the linear terms are). This motivated us to come up with a new notion of rank for non-idempotent intersection types, based on linear types (the ones derived in a linear type system – a substructural type system in which each assumption must be used exactly once, corresponding to the implicational fragment of linear logic [17]). The relation between non-idempotent intersection types and linearity was first introduced by Kfoury [20] and further explored by de Carvalho [12], who established its relation with linear logic.

Here we propose a new definition of rank for intersection types, which we call *linear rank* and differs from the classical one in the base case – instead of simple types, linear rank 0 intersection types are the linear types – and in the introduction of the functional type constructor "linear arrow" $\multimap$.

▶ **Definition 7** (Linear rank of intersection types)**.** *Let* $\mathbb{T}_{\mathbb{L}0} = \mathbb{V} \cup \{\tau_1 \multimap \tau_2 \mid \tau_1, \tau_2 \in \mathbb{T}_{\mathbb{L}0}\}$ *be the set of* ***linear types*** *and* $\mathbb{T}_{\mathbb{L}1} = \{\tau_1 \cap \cdots \cap \tau_m \mid \tau_1, \dots, \tau_m \in \mathbb{T}_{\mathbb{L}0}, m \geq 1\}$ *the set of sequences of linear types. The set* $\mathbb{T}_{\mathbb{L}k}$*, of* linear rank *k intersection types (for* $k \geq 2$*), can be defined recursively in the following way (*$n \geq 3$*,* $m \geq 2$*):*

$$\begin{aligned}\mathbb{T}_{\mathbb{L}2} &= \mathbb{T}_{\mathbb{L}0} \cup \{\tau \multimap \sigma \mid \tau \in \mathbb{T}_{\mathbb{L}0}, \sigma \in \mathbb{T}_{\mathbb{L}2}\} \\ &\quad \cup \{\tau_1 \cap \cdots \cap \tau_m \to \sigma \mid \tau_1, \dots, \tau_m \in \mathbb{T}_{\mathbb{L}0}, \sigma \in \mathbb{T}_{\mathbb{L}2}\} \\ \mathbb{T}_{\mathbb{L}n} &= \mathbb{T}_{\mathbb{L}n-1} \cup \{\rho \multimap \sigma \mid \rho \in \mathbb{T}_{\mathbb{L}n-1}, \sigma \in \mathbb{T}_{\mathbb{L}n}\} \\ &\quad \cup \{\rho_1 \cap \cdots \cap \rho_m \to \sigma \mid \rho_1, \dots, \rho_m \in \mathbb{T}_{\mathbb{L}n-1}, \sigma \in \mathbb{T}_{\mathbb{L}n}\}\end{aligned}$$

Initially, the idea for the change arose from our interest in using rank-restricted intersection types to estimate the number of evaluation steps of a λ-term while inferring its type. While defining the intersection type system to obtain quantitative information, we realized that the ranks could be potentially more useful for that purpose if the base case was changed to types that give more quantitative information in comparison to simple types, which is the case for linear types – for instance, if a term is typed with a linear rank 2 intersection type, one knows that each occurrence of its arguments is linear, meaning that they will be used exactly once.

The relation between the standard definition of rank and our definition of linear rank is not clear, and most likely non-trivial. Note that the set of terms typed using standard rank 2 intersection types [18, 26] and linear rank 2 intersection types is not the same. For instance, again, the term $(\lambda x.xx)(\lambda fx.f(fx))$, typable with a simple type in the standard Rank 2 Intersection Type System, is not typable in the Linear Rank 2 Intersection Type System, because, as the term $(\lambda fx.f(fx))$ is not linear and intersection is not idempotent, by Definition 7, the type of $(\lambda x.xx)(\lambda fx.f(fx))$ is now (linear) rank 3. This relation between rank and linear rank is an interesting question that will not be covered here, but one that we would like to explore in the future.

3.2 Type System

We now define a new type system for the λ-calculus with linear rank 2 non-idempotent intersection types.

▶ **Definition 8** (Substitution)**.** *Let* $S = [N/x]$ *denote a* substitution*. Then the result of substituting the term* N *for each free occurrence of* x *in the term* M*, denoted by* $M[N/x]$ *(or* $\mathcal{S}(M)$*), is inductively defined as follows:*

$$\begin{aligned}x[N/x] &= N; \\ x_1[N/x_2] &= x_1, \ \textit{if } x_1 \neq x_2; \\ (M_1M_2)[N/x] &= (M_1[N/x])(M_2[N/x]); \\ (\lambda x.M)[N/x] &= \lambda x.M; \\ (\lambda x_1.M)[N/x_2] &= \lambda x_1.(M[N/x_2]), \ \textit{if } x_1 \neq x_2.\end{aligned}$$

▶ **Notation.** *We write* $M[M_1/x_1, M_2/x_2, \dots, M_n/x_n]$ *for*
$(\dots((M[M_1/x_1])[M_2/x_2])\dots)[M_n/x_n]$.

Composing two substitutions $\mathcal{S}_1$ and $\mathcal{S}_2$ results in a substitution $\mathcal{S}_2 \circ \mathcal{S}_1$ that when applied, has the same effect as applying $\mathcal{S}_1$ followed by $\mathcal{S}_2$.

▶ **Definition 9** (Substitution composition). *The* composition *of two substitutions* $\mathcal{S}_1 = [N_1/x_1]$ *and* $\mathcal{S}_2 = [N_2/x_2]$*, denoted by* $\mathcal{S}_2 \circ \mathcal{S}_1$*, is defined as:*

$$\mathcal{S}_2 \circ \mathcal{S}_1(M) = M[N_1/x_1, N_2/x_2].$$

Also, we consider that the operation is right-associative:

$$\mathcal{S}_1 \circ \mathcal{S}_2 \circ \cdots \circ \mathcal{S}_{n-1} \circ \mathcal{S}_n = \mathcal{S}_1 \circ (\mathcal{S}_2 \circ \cdots \circ (\mathcal{S}_{n-1} \circ \mathcal{S}_n) \ldots).$$

▶ **Notation.** *From now on, we will use* α *to range over a countable infinite set* $\mathbb{V}$ *of type variables,* τ *to range over the set* $\mathbb{T}_{\mathbb{L}0}$ *of linear types,* $\vec{\tau}$ *to range over the set* $\mathbb{T}_{\mathbb{L}1}$ *of linear type sequences and* σ *to range over the set* $\mathbb{T}_{\mathbb{L}2}$ *of linear rank 2 intersection types. In all cases, we may use or not single quotes and/or number subscripts.*

▶ **Definition 10.**

- *A* statement *is an expression of the form* $M : \vec{\tau}$*, where* $\vec{\tau}$ *is called the* predicate*, and the term* M *is called the* subject *of the statement.*
- *A* declaration *is a statement where the subject is a term variable.*
- *The comma operator (,) appends a declaration to the end of a list (of declarations). The list* (Γ_1, Γ_2) *is the list that results from appending the list* Γ_2 *to the end of the list* Γ_1*.*
- *A finite list of declarations is* consistent *if and only if the term variables are all distinct.*
- *An* environment *is a consistent finite list of declarations which predicates are sequences of linear types (i.e., elements of* $\mathbb{T}_{\mathbb{L}1}$*) and we use* Γ *(possibly with single quotes and/or number subscripts) to range over environments.*
- *An environment* $\Gamma = [x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n]$ *induces a partial function* Γ *with domain* $\mathsf{dom}(\Gamma) = \{x_1, \ldots, x_n\}$*, and* $\Gamma(x_i) = \vec{\tau}_i$*.*
- *We write* Γ_x *for the resulting environment of eliminating the declaration of* x *from* Γ *(if there is no declaration of* x *in* Γ*, then* $\Gamma_x = \Gamma$*).*
- *We extend the notion of substitution to environments in the following way:*

  $$\mathcal{S}(\Gamma) = [\mathcal{S}(x_1) : \vec{\tau}_1, \ldots, \mathcal{S}(x_n) : \vec{\tau}_n] \qquad \text{if } \Gamma = [x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n]$$

- *We write* $\Gamma_1 \equiv \Gamma_2$ *if the environments* Γ_1 *and* Γ_2 *are equal up to the order of the declarations.*
- *If* Γ_1 *and* Γ_2 *are environments, the environment* $\Gamma_1 + \Gamma_2$ *is defined as follows: for each* $x \in \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$*,*

  $$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \notin \mathsf{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \notin \mathsf{dom}(\Gamma_1) \\ \Gamma_1(x) \cap \Gamma_2(x) & \text{otherwise} \end{cases}$$

  with the declarations of the variables in $\mathsf{dom}(\Gamma_1)$ *in the beginning of the list, by the same order they appear in* Γ_1*, followed by the declarations of the variables in* $\mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1)$*, by the order they appear in* Γ_2*.*

▶ **Definition 11** (Linear Rank 2 Intersection Type System). *In the Linear Rank 2 Intersection Type System, we say that* M *has type* σ *given the environment* Γ*, and write* $\Gamma \vdash_2 M : \sigma$*, if it can be obtained from the* derivation rules *in Figure 2.*

▶ **Example 12.** Let us write $\overset{\multimap}{\alpha}$ for the type $(\alpha \multimap \alpha)$. For the λ-term $(\lambda x.xx)(\lambda y.y)$, the following derivation is obtained:

$$\dfrac{\dfrac{\dfrac{\dfrac{[x_1 : \overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha}] \vdash_2 x_1 : \overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha} \qquad [x_2 : \overset{\multimap}{\alpha}] \vdash_2 x_2 : \overset{\multimap}{\alpha}}{[x_1 : \overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha}, x_2 : \overset{\multimap}{\alpha}] \vdash_2 x_1 x_2 : \overset{\multimap}{\alpha}}}{[x : (\overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha}) \cap \overset{\multimap}{\alpha}] \vdash_2 xx : \overset{\multimap}{\alpha}}}{[\,] \vdash_2 \lambda x.xx : (\overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha}) \cap \overset{\multimap}{\alpha} \to \overset{\multimap}{\alpha}} \qquad \dfrac{[y : \overset{\multimap}{\alpha}] \vdash_2 y : \overset{\multimap}{\alpha}}{[\,] \vdash_2 \lambda y.y : \overset{\multimap}{\alpha} \multimap \overset{\multimap}{\alpha}} \qquad \dfrac{[y : \alpha] \vdash_2 y : \alpha}{[\,] \vdash_2 \lambda y.y : \overset{\multimap}{\alpha}}}{[\,] \vdash_2 (\lambda x.xx)(\lambda y.y) : \overset{\multimap}{\alpha}}$$

$$[x : \tau] \vdash_2 x : \tau \qquad \text{(Axiom)}$$

$$\frac{\Gamma_1, x : \vec{\tau}_1, y : \vec{\tau}_2, \Gamma_2 \vdash_2 M : \sigma}{\Gamma_1, y : \vec{\tau}_2, x : \vec{\tau}_1, \Gamma_2 \vdash_2 M : \sigma} \qquad \text{(Exchange)}$$

$$\frac{\Gamma_1, x_1 : \vec{\tau}_1, x_2 : \vec{\tau}_2, \Gamma_2 \vdash_2 M : \sigma}{\Gamma_1, x : \vec{\tau}_1 \cap \vec{\tau}_2, \Gamma_2 \vdash_2 M[x/x_1, x/x_2] : \sigma} \qquad \text{(Contraction)}$$

$$\frac{\Gamma, x : \tau_1 \cap \cdots \cap \tau_n \vdash_2 M : \sigma \qquad n \geq 2}{\Gamma \vdash_2 \lambda x.M : \tau_1 \cap \cdots \cap \tau_n \to \sigma} \qquad (\to \text{Intro})$$

$$\frac{\Gamma \vdash_2 M_1 : \tau_1 \cap \cdots \cap \tau_n \to \sigma \qquad \Gamma_1 \vdash_2 M_2 : \tau_1 \;\cdots\; \Gamma_n \vdash_2 M_2 : \tau_n \qquad n \geq 2}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash_2 M_1 M_2 : \sigma} \quad (\to \text{Elim})$$

$$\frac{\Gamma, x : \tau \vdash_2 M : \sigma}{\Gamma \vdash_2 \lambda x.M : \tau \multimap \sigma} \qquad (\multimap \text{Intro})$$

$$\frac{\Gamma_1 \vdash_2 M_1 : \tau \multimap \sigma \qquad \Gamma_2 \vdash_2 M_2 : \tau}{\Gamma_1, \Gamma_2 \vdash_2 M_1 M_2 : \sigma} \qquad (\multimap \text{Elim})$$

Figure 2 Linear Rank 2 Intersection Type System.

3.3 Type Inference Algorithm

In this section we define a new type inference algorithm for the λ-calculus (Definition 23), which is sound (Theorem 32) and complete (Theorem 35) with respect to the Linear Rank 2 Intersection Type System.

Our algorithm is based on Trevor Jim's type inference algorithm [18] for a Rank 2 Intersection Type System that was introduced by Daniel Leivant in [21], where the algorithm was briefly covered. Different versions of the algorithm were later defined by Steffen van Bakel in [26] and by Trevor Jim in [18].

Part of the definitions, properties and proofs here presented are also adapted from [18].

▶ **Definition 13** (Type substitution). *Let $\mathbb{S} = [\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ denote a* type substitution, *where $\alpha_1, \ldots, \alpha_n$ are distinct type variables in $\mathbb{V}$ and $\tau_1, \ldots, \tau_n$ are types in $\mathbb{T}_{\mathbb{L}0}$.*

For any τ in $\mathbb{T}_{\mathbb{L}0}$, $\mathbb{S}(\tau) = \tau[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_n]$ is the type obtained by simultaneously substituting α_i by τ_i in τ, with $1 \leq i \leq n$.

The type $\mathbb{S}(\tau)$ is called an instance *of the type τ.*

The notion of type substitution can be extended to environments in the following way:

$$\mathbb{S}(\Gamma) = [x_1 : \mathbb{S}(\vec{\tau}_1), \ldots, x_n : \mathbb{S}(\vec{\tau}_n)] \qquad \textit{if } \Gamma = [x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n]$$

The environment $\mathbb{S}(\Gamma)$ is called an instance *of the environment Γ.*

If $\mathbb{S}_1 = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ *and* $\mathbb{S}_2 = [\tau'_1/\alpha'_1, \dots, \tau'_n/\alpha'_n]$ *are type substitutions such that the variables* $\alpha_1, \dots, \alpha_n, \alpha'_1, \dots, \alpha'_n$ *are all distinct, then the type substitution* $\mathbb{S}_1 \cup \mathbb{S}_2$ *is defined as* $\mathbb{S}_1 \cup \mathbb{S}_2 = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n, \tau'_1/\alpha'_1, \dots, \tau'_n/\alpha'_n]$.

Composing two type substitutions $\mathbb{S}_1$ and $\mathbb{S}_2$ results in a type substitution $\mathbb{S}_2 \circ \mathbb{S}_1$ that when applied, has the same effect as applying $\mathbb{S}_1$ followed by $\mathbb{S}_2$.

▶ **Definition 14** (Type substitution composition)**.** *The* composition *of two type substitutions* $\mathbb{S}_1 = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$ *and* $\mathbb{S}_2 = [\tau'_1/\alpha'_1, \dots, \tau'_m/\alpha'_m]$*, denoted by* $\mathbb{S}_2 \circ \mathbb{S}_1$*, is defined as:*

$$\mathbb{S}_2 \circ \mathbb{S}_1 = [\tau'_{i_1}/\alpha'_{i_1}, \dots, \tau'_{i_k}/\alpha'_{i_k}, \mathbb{S}_2(\tau_1)/\alpha_1, \dots, \mathbb{S}_2(\tau_n)/\alpha_n],$$

where $\{\alpha'_{i_1}, \dots, \alpha'_{i_k}\} = \{\alpha'_1, \dots, \alpha'_m\} \setminus \{\alpha_1, \dots, \alpha_n\}$.

Also, we consider that the operation is right-associative:

$$\mathbb{S}_1 \circ \mathbb{S}_2 \circ \dots \circ \mathbb{S}_{n-1} \circ \mathbb{S}_n = \mathbb{S}_1 \circ (\mathbb{S}_2 \circ \dots \circ (\mathbb{S}_{n-1} \circ \mathbb{S}_n) \dots).$$

3.3.1 Unification

We now recall Robinson's unification [24], for the special case of equations involving simple types. For the unification algorithm we follow a latter (more efficient) presentation by Martelli and Montanari [22].

▶ **Definition 15** (Unification problem)**.** *A* unification problem *is a finite set of equations* $P = \{\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n\}$*. A* unifier *(or* solution*) is a substitution* $\mathbb{S}$*, such that* $\mathbb{S}(\tau_i) = \mathbb{S}(\tau'_i)$*, for* $1 \le i \le n$*. We call* $\mathbb{S}(\tau_i)$ *(or* $\mathbb{S}(\tau'_i)$*) a* common instance *of* τ_i *and* τ'_i*.* P *is* unifiable *if it has at least one unifier.* $\mathcal{U}(P)$ *is the set of unifiers of* P*.*

▶ **Example 16.** The types $\alpha_1 \multimap \alpha_2 \multimap \alpha_1$ and $(\alpha_3 \multimap \alpha_3) \multimap \alpha_4$ are *unifiable.* For the type substitution $\mathbb{S} = [(\alpha_3 \multimap \alpha_3)/\alpha_1, (\alpha_2 \multimap (\alpha_3 \multimap \alpha_3))/\alpha_4]$, the *common instance* is $(\alpha_3 \multimap \alpha_3) \multimap \alpha_2 \multimap (\alpha_3 \multimap \alpha_3)$.

▶ **Definition 17** (Most general unifier)**.** *A substitution* $\mathbb{S}$ *is a* most general unifier *(MGU) of* P *if* $\mathbb{S}$ *is the least element of* $\mathcal{U}(P)$*. That is,*

$$\mathbb{S} \in \mathcal{U}(P) \text{ and } \forall \mathbb{S}_1 \in \mathcal{U}(P). \exists \mathbb{S}_2. \mathbb{S}_1 = \mathbb{S}_2 \circ \mathbb{S}.$$

▶ **Example 18.** Consider the types $\tau_1 = (\alpha_1 \multimap \alpha_1)$ and $\tau_2 = (\alpha_2 \multimap \alpha_3)$.

The type substitution $\mathbb{S}' = [(\alpha_4 \multimap \alpha_5)/\alpha_1, (\alpha_4 \multimap \alpha_5)/\alpha_2, (\alpha_4 \multimap \alpha_5)/\alpha_3]$ is a unifier of τ_1 and τ_2, but it is not the MGU.

The MGU of τ_1 and τ_2 is $\mathbb{S} = [\alpha_3/\alpha_1, \alpha_3/\alpha_2]$. The common instance of τ_1 and τ_2 by $\mathbb{S}'$, $(\alpha_4 \multimap \alpha_5) \multimap (\alpha_4 \multimap \alpha_5)$, is an instance of $(\alpha_3 \multimap \alpha_3)$, the common instance by $\mathbb{S}$.

▶ **Definition 19** (Solved form)**.** *A unification problem* $P = \{\alpha_1 = \tau_1, \dots, \alpha_n = \tau_n\}$ *is in* solved form *if* $\alpha_1, \dots, \alpha_n$ *are all pairwise distinct variables that do not occur in any of the* τ_i*. In this case, we define* $\mathbb{S}_P = [\tau_1/\alpha_1, \dots, \tau_n/\alpha_n]$*.*

▶ **Definition 20** (Type unification)**.** *We define the following relation* $\Rightarrow$ *on type unification problems (for types in* $\mathbb{T}_{\mathbb{L}0}$*):*

$$\begin{array}{lll}
\{\tau = \tau\} \cup P & \Rightarrow & P \\
\{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P & \Rightarrow & \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P \\
\{\tau_1 \multimap \tau_2 = \alpha\} \cup P & \Rightarrow & \{\alpha = \tau_1 \multimap \tau_2\} \cup P \\
\{\alpha = \tau\} \cup P & \Rightarrow & \{\alpha = \tau\} \cup P[\tau/\alpha] \qquad \textit{if } \alpha \in \mathsf{fv}(P) \setminus \mathsf{fv}(\tau) \\
\{\alpha = \tau\} \cup P & \Rightarrow & \mathsf{FAIL} \qquad \textit{if } \alpha \in \mathsf{fv}(\tau) \textit{ and } \alpha \neq \tau
\end{array}$$

where $P[\tau/\alpha]$ *corresponds to the notion of type substitution extended to type unification problems. If* $P = \{\tau_1 = \tau'_1, \dots, \tau_n = \tau'_n\}$*, then* $P[\tau/\alpha] = \{\tau_1[\tau/\alpha] = \tau'_1[\tau/\alpha], \dots, \tau_n[\tau/\alpha] = \tau'_n[\tau/\alpha]\}$*. And* $\mathsf{fv}(P)$ *and* $\mathsf{fv}(\tau)$ *are the sets of free type variables in* P *and* τ*, respectively. Since in our system all occurrences of type variables are free,* $\mathsf{fv}(P)$ *and* $\mathsf{fv}(\tau)$ *are the sets of type variables in* P *and* τ*, respectively.*

▶ **Definition 21** (Unification algorithm). *Let* P *be a unification problem (with types in* $\mathbb{T}_{\mathbb{L}0}$*). The unification function* UNIFY(P) *that decides whether* P *has a solution and, if so, returns the MGU of* P *(see [24]), is defined as:*

```
function UNIFY($P$)
    while $P \Rightarrow P'$ do
        $P := P'$;
    if $P$ is in solved form then
        return $\mathbb{S}_P$;
    else
        FAIL;
```

▶ **Example 22.** Consider again the types $\alpha_1 \multimap \alpha_1$ and $\alpha_2 \multimap \alpha_3$ in Example 18. For the unification problem $P = \{\alpha_1 \multimap \alpha_1 = \alpha_2 \multimap \alpha_3\}$, UNIFY$(P)$ performs the following transformations over P:

$$\begin{aligned}
\{\alpha_1 \multimap \alpha_1 = \alpha_2 \multimap \alpha_3\} &\Rightarrow \{\alpha_1 = \alpha_2, \alpha_1 = \alpha_3\} \cup \{\ \} &&= \{\alpha_1 = \alpha_2, \alpha_1 = \alpha_3\}\\
&\Rightarrow \{\alpha_1 = \alpha_2\} \cup \{\alpha_1 = \alpha_3\}[\alpha_2/\alpha_1] &&= \{\alpha_1 = \alpha_2, \alpha_2 = \alpha_3\}\\
&\Rightarrow \{\alpha_2 = \alpha_3\} \cup \{\alpha_1 = \alpha_2\}[\alpha_3/\alpha_2] &&= \{\alpha_1 = \alpha_3, \alpha_2 = \alpha_3\}
\end{aligned}$$

and, since $\{\alpha_1 = \alpha_3, \alpha_2 = \alpha_3\}$ is in solved form, it returns the type substitution $[\alpha_3/\alpha_1, \alpha_3/\alpha_2]$.

3.3.2 Type Inference

▶ **Definition 23** (Type inference algorithm). *Let* Γ *be an environment,* M *a* λ*-term,* σ *a linear rank 2 intersection type and* UNIFY *the function in Definition 21. The function* $\mathsf{T}(M) = (\Gamma, \sigma)$ *defines a type inference algorithm for the* λ*-calculus in the Linear Rank 2 Intersection Type System, in the following way:*

1. *If* $M = x$*, then* $\Gamma = [x : \alpha]$ *and* $\sigma = \alpha$*, where* α *is a new variable;*
2. *If* $M = \lambda x.M_1$ *and* $\mathsf{T}(M_1) = (\Gamma_1, \sigma_1)$ *then:*
   a. *if* $x \notin \mathsf{dom}(\Gamma_1)$*, then* FAIL*;*
   b. *if* $(x : \tau) \in \Gamma_1$*, then* $\mathsf{T}(M) = (\Gamma_{1x}, \tau \multimap \sigma_1)$*;*
   c. *if* $(x : \tau_1 \cap \dots \cap \tau_n) \in \Gamma_1$ *(with* $n \geq 2$*), then* $\mathsf{T}(M) = (\Gamma_{1x}, \tau_1 \cap \dots \cap \tau_n \to \sigma_1)$*.*
3. *If* $M = M_1M_2$*, then:*
   a. *if* $\mathsf{T}(M_1) = (\Gamma_1, \alpha_1)$ *and* $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$*,*
   then $\mathsf{T}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3))$*,*
   where $\mathbb{S} =$ UNIFY$(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ *and* α_2, α_3 *are new variables;*
   b. *if* $\mathsf{T}(M_1) = (\Gamma'_1, \tau'_1 \cap \dots \cap \tau'_n \to \sigma'_1)$ *(with* $n \geq 2$*) and, for each* $1 \leq i \leq n$*,*
   $\mathsf{T}(M_2) = (\Gamma_i, \tau_i)^{\mathrm{A}}$*,*
   then $\mathsf{T}(M) = (\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1))$*,*
   where $\mathbb{S} =$ UNIFY$(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$*;*

[A] Note that Γ_i, τ_i can all be different up to renaming of variables.

c. *<u>if</u>* $\mathsf{T}(M_1) = (\Gamma_1, \tau \multimap \sigma_1)$ *and* $\mathsf{T}(M_2) = (\Gamma_2, \tau_2)$,
<u>then</u> $\mathsf{T}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1))$,
where $\mathbb{S} = \mathsf{UNIFY}(\{\tau_2 = \tau\})$*;*

d. *<u>otherwise</u>* FAIL.

▶ **Example 24.** Let us show the type inference process for the λ-term $\lambda x.xx$.

- By rule 1., $\mathsf{T}(x) = ([x : \alpha_1], \alpha_1)$.
- By rule 1., again, $\mathsf{T}(x) = ([x : \alpha_2], \alpha_2)$.
- Then by rule 3.(a), $\mathsf{T}(xx) = (\mathbb{S}([x : \alpha_1] + [x : \alpha_2]), \mathbb{S}(\alpha_4)) = (\mathbb{S}([x : \alpha_1 \cap \alpha_2]), \mathbb{S}(\alpha_4))$, where $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 = \alpha_3\}) = [\alpha_3 \multimap \alpha_4/\alpha_1, \alpha_3/\alpha_2]$.
So $\mathsf{T}(xx) = ([x : (\alpha_3 \multimap \alpha_4) \cap \alpha_3], \alpha_4)$.
- Finally, by rule 2.(c), $\mathsf{T}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4)$.

▶ **Example 25.** Let us now show the type inference process for the λ-term $(\lambda x.xx)(\lambda y.y)$.

- From the previous example, we have $\mathsf{T}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4)$.
- By rules 1. and 2.(b), for the identity, the algorithm gives $\mathsf{T}(\lambda y.y) = ([\,], \alpha_1 \multimap \alpha_1)$.
- By rules 1. and 2.(b), again, for the identity, $\mathsf{T}(\lambda y.y) = ([\,], \alpha_2 \multimap \alpha_2)$.
- Then by rule 3.(b), $\mathsf{T}((\lambda x.xx)(\lambda y.y)) = (\mathbb{S}([\,] + [\,] + [\,]), \mathbb{S}(\alpha_4)) = ([\,], \mathbb{S}(\alpha_4))$, where $\mathbb{S} = \mathsf{UNIFY}(\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\})$, calculated by performing the following transformations:

$$\begin{aligned}
\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\} &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_1 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\
&\Rightarrow \{\alpha_1 = \alpha_3, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\
&\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_4\}\\
&\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_4 = \alpha_2 \multimap \alpha_2\}\\
&\Rightarrow \{\alpha_1 = \alpha_2 \multimap \alpha_2, \alpha_3 = \alpha_2 \multimap \alpha_2, \alpha_4 = \alpha_2 \multimap \alpha_2\}
\end{aligned}$$

So $\mathbb{S} = [(\alpha_2 \multimap \alpha_2)/\alpha_1, (\alpha_2 \multimap \alpha_2)/\alpha_3, (\alpha_2 \multimap \alpha_2)/\alpha_4]$
and $\mathsf{T}((\lambda x.xx)(\lambda y.y)) = ([\,], \alpha_2 \multimap \alpha_2)$.

Now we show several properties of our type system and type inference algorithm, in order to prove the soundness and completeness of the algorithm with respect to the system.

▶ **Notation.** *We write* $\Phi \rhd \Gamma \vdash_2 M : \sigma$ *to denote that* Φ *is a derivation tree ending with* $\Gamma \vdash_2 M : \sigma$*. In this case,* $|\Phi|$ *is the depth of the derivation tree* Φ*.*

▶ **Lemma 26** (Substitution). *If* $\Phi \rhd \Gamma \vdash_2 M : \sigma$*, then* $\mathbb{S}(\Gamma) \vdash_2 M : \mathbb{S}(\sigma)$ *for any substitution* $\mathbb{S}$*.*

▶ **Lemma 27** (Relevance). *If* $\Phi \rhd \Gamma \vdash_2 M : \sigma$*, then* $x \in \mathsf{dom}(\Gamma)$ *if and only if* $x \in \mathsf{FV}(M)$*.*

▶ **Lemma 28.** *If* $\mathsf{T}(M) = (\Gamma, \sigma)$*, then* $x \in \mathsf{dom}(\Gamma)$ *if and only if* $x \in \mathsf{FV}(M)$*.*

▶ **Corollary 29.** *From Lemma 27 and Lemma 28, it follows that if* $\mathsf{T}(M) = (\Gamma, \sigma)$ *and* $\Gamma' \vdash_2 M : \sigma'$*, then* $\mathsf{dom}(\Gamma) = \mathsf{dom}(\Gamma')$*.*

▶ **Lemma 30.** *If* $\Phi_1 \rhd \Gamma \vdash_2 M : \sigma$*,* $x \in \mathsf{FV}(M)$ *and* y *does not occur in* M*, then there exists* $\Phi_2 \rhd \Gamma[y/x] \vdash_2 M[y/x] : \sigma$*, with* $|\Phi_1| = |\Phi_2|$*.*

▶ **Corollary 31.** *From Lemma 30, it follows that if* $\Gamma \vdash_2 M : \sigma$*,* $\{x_1, \ldots, x_n\} \subseteq \mathsf{FV}(M)$ *and* $y_1, \ldots, y_n$ *are all different variables not occurring in* M*, then* $\Gamma[y_1/x_1, \ldots, y_n/x_n] \vdash_2 M[y_1/x_1, \ldots, y_n/x_n] : \sigma$*.*

▶ **Theorem 32** (Soundness). *If* $\mathsf{T}(M) = (\Gamma, \sigma)$, *then* $\Gamma \vdash_2 M : \sigma$.

▶ **Lemma 33.** *If* $\mathsf{T}(M) = (\Gamma, \sigma)$, $x \in \mathsf{FV}(M)$ *and* y *does not occur in* M, *then* $\mathsf{T}(M[y/x]) = (\Gamma[y/x], \sigma)$.

▶ **Lemma 34.** *If* $\mathsf{T}(M) = (\Gamma, \sigma)$, *with* $\Gamma \equiv (\Gamma', y_1 : \vec{\tau}_1, y_2 : \vec{\tau}_2)$, *and* y *does not occur in* M, *then* $\mathsf{T}(M[y/y_1, y/y_2]) = (\Gamma'', \sigma)$, *with* $\Gamma'' \equiv (\Gamma', y : \vec{\tau}_1 \cap \vec{\tau}_2)$.

▶ **Theorem 35** (Completeness). *If* $\Phi \rhd \Gamma \vdash_2 M : \sigma$, *then* $\mathsf{T}(M) = (\Gamma', \sigma')$ *(for some environment* Γ' *and type* σ'*) and there is a substitution* $\mathbb{S}$ *such that* $\mathbb{S}(\sigma') = \sigma$ *and* $\mathbb{S}(\Gamma') \equiv \Gamma$.

Hence, we end up with a sound and complete type inference algorithm for the Linear Rank 2 Intersection Type System.

3.4 Remarks

A λ-term M is called a λI-term if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N at least once. Note that our type system and type inference algorithm only type λI-terms, but we could have extended them for the *affine terms* – a λ-term M is affine if and only if, for each subterm of the form $\lambda x.N$ in M, x occurs free in N at most once, and if each free variable of M has just one occurrence free in M.

There is no unique and final way of typing affine terms. For instance, in the systems in [1], arguments that do not occur in the body of the function get the empty type $[\,]$. Since we do not allow the empty sequence in our definition and adding it would make the system more complex, we decided to only work with λI-terms.

Regarding our choice of defining environments as lists and having the rules (Exchange) and (Contraction) in the type system, instead of defining environments as sets and using the $(+)$ operation for concatenation, that decision had to do with the fact that, this way, the system is closer to a linear type system. In the Linear Rank 2 Intersection Type System, a term is linear until we need to contract variables, so using these definitions makes us have more control over linearity and non-linearity. Also, it makes the system more easily extensible for other algebraic properties of intersection. We could also have rewritten the rule ($\to$ Elim) in order not to use the $(+)$ operation, which is something we might do in the future.

The downside of choosing these definitions is that it makes the proofs (in Section 3 and Section 4) more complex, as they are not syntax directed because of the rules (Exchange) and (Contraction).

4 Resource Inference

Given the quantitative properties of the Linear Rank 2 Intersection Types, we now aim to redefine the type system and the type inference algorithm, in order to infer not only the type of a λ-term, but also parameters related to resource usage. In this case, we are interested in obtaining the number of evaluation steps of the λ-term to its normal form, for the leftmost-outermost strategy.

4.1 Type System

The new type system defined in this chapter results from an adaptation and merge between our Linear Rank 2 Intersection Type System (Definition 11) and the system for the leftmost-outermost evaluation strategy presented in [1], as that system is able to derive a measure related to the number of evaluation steps for the leftmost-outermost strategy. We then begin by adapting some definitions from [1] and others that were already introduced in Section 3.

The predicates normal and neutral defining, respectively, the leftmost-outermost normal terms and neutral terms, are in Definition 36. The predicate $\mathsf{abs}(M)$ is true if and only if M is an abstraction; $\mathsf{normal}(\mathrm{M})$ means that M is in normal form; and $\mathsf{neutral}(M)$ means that M is in normal form and can never behave as an abstraction, i.e., it does not create a redex when applied to an argument.

▶ **Definition 36** (Leftmost-outermost normal forms).

$$\frac{}{\mathsf{neutral}(x)} \qquad \frac{\mathsf{neutral}(M) \qquad \mathsf{normal}(N)}{\mathsf{neutral}(MN)} \qquad \frac{\mathsf{neutral}(M)}{\mathsf{normal}(M)} \qquad \frac{\mathsf{normal}(M)}{\mathsf{normal}(\lambda x.M)}$$

▶ **Definition 37** (Leftmost-outermost evaluation strategy).

$$\frac{}{(\lambda x.M)N \longrightarrow M[N/x]} \qquad \frac{M \longrightarrow M'}{\lambda x.M \longrightarrow \lambda x.M'} \qquad \frac{M \longrightarrow M' \qquad \neg\mathsf{abs}(M)}{MN \longrightarrow M'N}$$

$$\frac{\mathsf{neutral}(N) \qquad M \longrightarrow M'}{NM \longrightarrow NM'}$$

▶ **Definition 38** (Finite rank multi-types). *We define the finite rank multi-types by the following grammar:*

$$\mathsf{tight} ::= \mathsf{Neutral} \mid \mathsf{Abs} \qquad \text{(Tight constants)}$$

$$t ::= \mathsf{tight} \mid \alpha \mid t \multimap t \qquad \text{(Rank 0 multi-types)}$$

$$\vec{t} ::= t \mid \vec{t} \cap \vec{t} \qquad \text{(Rank 1 multi-types)}$$

$$s ::= t \mid \vec{t} \rightarrow s \qquad \text{(Rank 2 multi-types)}$$

▶ **Definition 39.**

- *Here, a* statement *is an expression of the form $M : (\vec{\tau}, \vec{t})$, where the pair $(\vec{\tau}, \vec{t})$ is called the* predicate, *and the term M is called the* subject *of the statement.*
- *A* declaration *is a statement where the subject is a term variable.*
- *The comma operator (,) appends a declaration to the end of a list (of declarations). The list (Γ_1, Γ_2) is the list that results from appending the list Γ_2 to the end of the list Γ_1.*
- *A finite list of declarations is* consistent *if and only if the term variables are all distinct.*
- *An* environment *is a consistent finite list of declarations which predicates are pairs with a sequence from $\mathbb{T}_{\mathbb{L}1}$ as the first element and a rank 1 multi-type as the second element of the pair (i.e., the declarations are of the form $x : (\vec{\tau}, \vec{t})$), and we use Γ (possibly with single quotes and/or number subscripts) to range over environments.*
- *An environment $\Gamma = [x_1 : (\vec{\tau}_1, \vec{t}_1), \dots, x_n : (\vec{\tau}_n, \vec{t}_n)]$ induces a partial function Γ with domain $\mathsf{dom}(\Gamma) = \{x_1, \dots, x_n\}$, and $\Gamma(x_i) = (\vec{\tau}_i, \vec{t}_i)$.*
- *We write Γ_x for the resulting environment of eliminating the declaration of x from Γ (if there is no declaration of x in Γ, then $\Gamma_x = \Gamma$).*
- *We write $\Gamma_1 \equiv \Gamma_2$ if the environments Γ_1 and Γ_2 are equal up to the order of the declarations.*
- *If Γ_1 and Γ_2 are environments, the environment $\Gamma_1 + \Gamma_2$ is defined as follows: for each $x \in \mathsf{dom}(\Gamma_1) \cup \mathsf{dom}(\Gamma_2)$,*

$$(\Gamma_1 + \Gamma_2)(x) = \begin{cases} \Gamma_1(x) & \text{if } x \notin \mathsf{dom}(\Gamma_2) \\ \Gamma_2(x) & \text{if } x \notin \mathsf{dom}(\Gamma_1) \\ (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2) & \text{if } \Gamma_1(x) = (\vec{\tau}_1, \vec{t}_1) \text{ and } \Gamma_2(x) = (\vec{\tau}_2, \vec{t}_2) \end{cases}$$

  with the declarations of the variables in $\mathsf{dom}(\Gamma_1)$ in the beginning of the list, by the same order they appear in Γ_1, followed by the declarations of the variables in $\mathsf{dom}(\Gamma_2) \setminus \mathsf{dom}(\Gamma_1)$, by the order they appear in Γ_2.

- *We write* $\mathsf{tight}(s)$ *if* s *is of the form* tight *and* $\mathsf{tight}(t_1\cap\cdots\cap t_n)$ *if* $\mathsf{tight}(t_i)$ *for all* $1 \le i \le n$. *For* $\Gamma = [x_1 : (\vec{\tau_1}, \vec{t_1}), \ldots, x_n : (\vec{\tau_n}, \vec{t_n})]$, *we write* $\mathsf{tight}(\Gamma)$ *if* $\mathsf{tight}(\vec{t_i})$ *for all* $1 \le i \le n$, *in which case we also say that* Γ *is tight.*

▶ **Definition 40** (Linear Rank 2 Quantitative Type System). *In the Linear Rank 2 Quantitative Type System, we say that* M *has type* σ *and multi-type* s *given the environment* Γ, *with index* b, *and write* $\Gamma \vdash^b M : (\sigma, s)$, *if it can be obtained from the* derivation rules *in Figure 3.*

The tight rules (the t-indexed ones) are used to introduce the tight constants Neutral and Abs, and they are related to minimal typings. Note that the index is only incremented in rules ($\multimap$ Intro) and ($\to$ Intro), as these are used to type abstractions that will be applied, contrary to the abstractions typed with the constant Abs.

▶ **Notation.** *We write* $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ *if* Φ *is a derivation tree ending with* $\Gamma \vdash^b M : (\sigma, s)$. *In this case,* $|\Phi|$ *is the depth of the derivation tree* Φ.

▶ **Definition 41** (Tight derivations). *A derivation* $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ *is tight if* $\mathsf{tight}(s)$ *and* $\mathsf{tight}(\Gamma)$.

Similarly to what has been done in [1], in this section we prove that, in the Linear Rank 2 Quantitative Type System, whenever a term is tightly typable with index b, then b is exactly the number of evaluations steps to leftmost-outermost normal form.

▶ **Example 42.** Let $M = (\lambda x_1.(\lambda x_2.x_2x_1)x_1)I$, where I is the identity function $\lambda y.y$.

Let us first consider the leftmost-outermost evaluation of M to normal form:

$$(\lambda x_1.(\lambda x_2.x_2x_1)x_1)I \longrightarrow (\lambda x_2.x_2I)I \longrightarrow II \longrightarrow I$$

So the evaluation sequence has length 3.

Let us write $\overset{\multimap}{\alpha}$ for the type $(\alpha \multimap \alpha)$ and $\overset{\multimap}{\mathsf{Abs}}$ for the type $\mathsf{Abs} \multimap \mathsf{Abs}$.

To make the derivation tree easier to read, let us first get the following derivation Φ for the term $\lambda x_1.(\lambda x_2.x_2x_1)x_1$:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{[x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^0 x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}) \qquad [x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}{[x_2 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}), x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 x_2x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}}{[x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^1 \lambda x_2.x_2x_3 : ((\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}) \multimap \overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}\multimap \mathsf{Abs})} \qquad [x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^0 x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}}))}{[x_3 : (\overset{\multimap}{\alpha}, \mathsf{Abs}), x_4 : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})] \vdash^1 (\lambda x_2.x_2x_3)x_4 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}}{[x_1 : (\overset{\multimap}{\alpha} \cap (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}), \mathsf{Abs} \cap \overset{\multimap}{\mathsf{Abs}})] \vdash^1 (\lambda x_2.x_2x_1)x_1 : (\overset{\multimap}{\alpha}, \mathsf{Abs})}}{[\,] \vdash^2 \lambda x_1.(\lambda x_2.x_2x_1)x_1 : ((\overset{\multimap}{\alpha} \cap (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha})) \to \overset{\multimap}{\alpha}, (\mathsf{Abs} \cap \overset{\multimap}{\mathsf{Abs}}) \to \mathsf{Abs})}$$

Then for the λ-term M, the following tight derivation is obtained:

$$\dfrac{\Phi \qquad \dfrac{[y : (\alpha, \mathsf{Neutral})] \vdash^0 y : (\alpha, \mathsf{Neutral})}{[\,] \vdash^0 I : (\overset{\multimap}{\alpha}, \mathsf{Abs})} \qquad \dfrac{[y : (\overset{\multimap}{\alpha}, \mathsf{Abs})] \vdash^0 y : (\overset{\multimap}{\alpha}, \mathsf{Abs})}{[\,] \vdash^1 I : (\overset{\multimap}{\alpha}\multimap\overset{\multimap}{\alpha}, \overset{\multimap}{\mathsf{Abs}})}}{[\,] \vdash^3 (\lambda x_1.(\lambda x_2.x_2x_1)x_1)I : (\overset{\multimap}{\alpha}, \mathsf{Abs})}$$

So indeed, the index 3 represents the number of evaluation steps to leftmost-outermost normal form.

We now show several properties of the type system, adapted from [1], in order to prove the *tight correctness* (Theorem 49).

▶ **Lemma 43** (Tight spreading on neutral terms). *If* M *is a term such that* $\mathsf{neutral}(M)$ *and* $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ *is a typing derivation such that* $\mathsf{tight}(\Gamma)$, *then* $\mathsf{tight}(s)$.

$$[x : (\tau, t)] \vdash^{0} x : (\tau, t) \qquad \text{(Axiom)}$$

$$\frac{\Gamma_1, x : (\vec{\tau}_1, \vec{t}_1), y : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b} M : (\sigma, s)}{\Gamma_1, y : (\vec{\tau}_2, \vec{t}_2), x : (\vec{\tau}_1, \vec{t}_1), \Gamma_2 \vdash^{b} M : (\sigma, s)} \qquad \text{(Exchange)}$$

$$\frac{\Gamma_1, x_1 : (\vec{\tau}_1, \vec{t}_1), x_2 : (\vec{\tau}_2, \vec{t}_2), \Gamma_2 \vdash^{b} M : (\sigma, s)}{\Gamma_1, x : (\vec{\tau}_1 \cap \vec{\tau}_2, \vec{t}_1 \cap \vec{t}_2), \Gamma_2 \vdash^{b} M[x/x_1, x/x_2] : (\sigma, s)} \qquad \text{(Contraction)}$$

$$\frac{\Gamma, x : (\tau, t) \vdash^{b} M : (\sigma, s)}{\Gamma \vdash^{b+1} \lambda x.M : (\tau \multimap \sigma, t \multimap s)} \qquad (\multimap \text{ Intro})$$

$$\frac{\Gamma, x : (\tau, \mathsf{tight}) \vdash^{b} M : (\sigma, \mathsf{tight})}{\Gamma \vdash^{b} \lambda x.M : (\tau \multimap \sigma, \mathsf{Abs})} \qquad (\multimap \text{ Intro}_{\mathrm{t}})$$

$$\frac{\Gamma, x : (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n) \vdash^{b} M : (\sigma, s) \qquad n \geq 2}{\Gamma \vdash^{b+1} \lambda x.M : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, t_1 \cap \cdots \cap t_n \to s)} \qquad (\to \text{ Intro})$$

$$\frac{\Gamma, x : (\tau_1 \cap \cdots \cap \tau_n, \vec{t}) \vdash^{b} M : (\sigma, \mathsf{tight}) \qquad \mathsf{tight}(\vec{t}) \qquad n \geq 2}{\Gamma \vdash^{b} \lambda x.M : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, \mathsf{Abs})} \qquad (\to \text{ Intro}_{\mathrm{t}})$$

$$\frac{\Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, t \multimap s) \qquad \Gamma_2 \vdash^{b_2} M_2 : (\tau, t)}{\Gamma_1, \Gamma_2 \vdash^{b_1 + b_2} M_1 M_2 : (\sigma, s)} \qquad (\multimap \text{ Elim})$$

$$\frac{\Gamma_1 \vdash^{b_1} M_1 : (\tau \multimap \sigma, \mathsf{Neutral}) \qquad \Gamma_2 \vdash^{b_2} M_2 : (\tau, \mathsf{tight})}{\Gamma_1, \Gamma_2 \vdash^{b_1 + b_2} M_1 M_2 : (\sigma, \mathsf{Neutral})} \qquad (\multimap \text{ Elim}_{\mathrm{t}})$$

$$\frac{\begin{array}{l}\Gamma \vdash^{b} M_1 : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, t_1 \cap \cdots \cap t_n \to s) \\ \Gamma_1 \vdash^{b_1} M_2 : (\tau_1, t_1) \ \cdots \ \Gamma_n \vdash^{b_n} M_2 : (\tau_n, t_n) \qquad n \geq 2\end{array}}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash^{b + b_1 + \cdots + b_n} M_1 M_2 : (\sigma, s)} \qquad (\to \text{ Elim})$$

$$\frac{\begin{array}{c}\Gamma \vdash^{b} M_1 : (\tau_1 \cap \cdots \cap \tau_n \to \sigma, \mathsf{Neutral}) \\ \Gamma_1 \vdash^{b_1} M_2 : (\tau_1, \mathsf{tight}) \ \cdots \ \Gamma_n \vdash^{b_n} M_2 : (\tau_n, \mathsf{tight}) \qquad n \geq 2\end{array}}{\Gamma, \sum_{i=1}^{n} \Gamma_i \vdash^{b + b_1 + \cdots + b_n} M_1 M_2 : (\sigma, \mathsf{Neutral})} \qquad (\to \text{ Elim}_{\mathrm{t}})$$

Figure 3 Linear Rank 2 Quantitative Type System.

▶ **Lemma 44** (Properties of tight typings for normal forms)**.** *Let M be such that* normal(M) *and $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ be a typing derivation.*

(i) Tightness*: if Φ is tight, then $b = 0$.*

(ii) Neutrality*: if $s =$* Neutral *then* neutral(M)*.*

▶ **Lemma 45** (Relevance)**.** *If $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$, then $x \in$* dom(Γ) *if and only if $x \in$* FV(M)*.*

▶ **Lemma 46** (Substitution and typings)**.** *Let $\Phi \rhd \Gamma \vdash^b M_1 : (\sigma, s)$ be a derivation with $x \in$* dom(Γ) *and $\Gamma(x) = (\tau_1 \cap \cdots \cap \tau_n, t_1 \cap \cdots \cap t_n)$, for $n \geq 1$. And, for each $1 \leq i \leq n$, let $\Phi_i \rhd \Gamma_i \vdash^{b_i} M_2 : (\tau_i, t_i)$.*

Then there exists a derivation $\Phi' \rhd \Gamma_x, \sum_{i=1}^n \Gamma_i \vdash^{b+b_1+\cdots+b_n} M_1[M_2/x] : (\sigma, s)$. Moreover, if the derivations $\Phi, \Phi_1, \ldots, \Phi_n$ are tight, then so is the derivation Φ'.

Proof (Sketch). The proof is by induction on $|\Phi|$. In fact, without loss of generality, we assume that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$, so that $\Gamma_x, \sum_{i=1}^n \Gamma_i$ is consistent. Otherwise, we could simply rename the free variables in M_1 to get M_1' (and the same derivation Φ, with the variables renamed) such that $\mathsf{FV}(M_1') \cap \mathsf{FV}(M_2) = \emptyset$. Then, our proof of the lemma considers M_1, M_2 such that $\mathsf{FV}(M_1) \cap \mathsf{FV}(M_2) = \emptyset$, obtaining a derivation Φ' (with the renamed variables) and finally we could apply the rule (Contraction) (and (Exchange), when necessary) to the variables that were renamed in M_1, in order to end up with the more general form of the derivation. ◀

We now show an important property that relates contracted terms with their linear counterpart. Basically, it says that the following diagram commutes (under the described conditions):

$$\begin{tikzcd}
M' \arrow[r, "\beta"] \arrow[d, "\mathcal{S}(M')"'] & N' \arrow[d, "\mathcal{S}(N')"] \\
M \arrow[r, "\beta"'] & N
\end{tikzcd}$$

▶ **Lemma 47.** *Let $M \longrightarrow N$ and $M = \mathcal{S}(M')$ for some substitution $\mathcal{S} = [x/x_1, x/x_2]$ where x_1, x_2 occur free in M' and x does not occur in M'. Then there exists a term N' such that $N = \mathcal{S}(N')$ and $M' \longrightarrow N'$.*

▶ **Convention 4.1.** *Without loss of generality, we assume that, in a derivation tree, all contracted variables (i.e., variables that, at some point in the derivation tree, disappear from the term and environment by an application of the (Contraction) rule) are different from any other variable in the derivation tree.*

We also assume that when applying (Contraction), the new variables that substitute the contracted ones are also different from any other variable in the derivation tree.

▶ **Lemma 48** (Quantitative subject reduction)**.** *If $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ is tight and $M \longrightarrow N$, then $b \geq 1$ and there exists a tight derivation Φ' such that $\Phi' \rhd \Gamma \vdash^{b-1} N : (\sigma, s)$.*

Proof (Sketch). We prove the following stronger statement:

If $M \longrightarrow N$, $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$, tight$(\Gamma)$, and either tight$(s)$ or $\neg$abs(M), then there exists a derivation $\Phi' \rhd \Gamma \vdash^{b-1} N : (\sigma, s)$.

The proof of this statement follows by induction on $M \longrightarrow N$. The complexity of this proof is related to the (Exchange) and (Contraction) rules. Since these rules are not syntax-directed, we cannot use M do decide which rule was last applied in the derivation, since (Exchange) and (Contraction) rules can always be the last rule applied. The proof is therefore more complex and requires the use of Convention 4.1. ◀

▶ **Theorem 49** (Tight correctness). *If $\Phi \rhd \Gamma \vdash^b M : (\sigma, s)$ is a tight derivation, then there exists N such that $M \longrightarrow^b N$ and* $\mathsf{normal}(N)$. *Moreover, if $s = \mathsf{Neutral}$ then* $\mathsf{neutral}(N)$.

4.2 Type Inference Algorithm

We now extend the type inference algorithm defined in Section 3 (Definition 23) to also infer the number of reduction steps of the typed term to its normal form, when using the leftmost-outermost evaluation strategy.

This is done by slightly modifying the unification algorithm in Definition 21 and the algorithm in Definition 23, which will now carry and update a measure b that relates to the number of reduction steps. First, recall Definition 20, presented in Section 3.

▶ **Definition 50** (Quantitative Unification Algorithm). *Let P be a unification problem (with types in $\mathbb{T}_{\mathbb{L}0}$). The new unification function $\mathsf{UNIFY_Q}(P)$, which decides whether P has a solution and, if so, returns the MGU of P and an integer b used for counting purposes in the inference algorithm, is defined as:*

```
function UNIFY_Q(P)
    $b := 0$;
    while $P \Rightarrow P'$ do
        if $P = \{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P_1$ and $P' = \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P_1$ then
            $b := b + 1$;
        $P := P'$;
    if $P$ is in solved form then
        return $(\mathbb{S}_P, b)$;
    else
        FAIL;
```

Let $\mathbb{T}_{\mathbb{L}1}$-environment be an environment as defined in Section 3, i.e., just like the definition we use in the current chapter, but the predicates are only the first element of the pair (i.e., a sequence from $\mathbb{T}_{\mathbb{L}1}$).

▶ **Definition 51** (Quantitative Type Inference Algorithm). *Let Γ be a $\mathbb{T}_{\mathbb{L}1}$-environment, M a λ-term, σ a linear rank 2 intersection type, b a quantitative measure and $\mathsf{UNIFY_Q}$ the function in Definition 50. The function $\mathsf{T_Q}(M) = (\Gamma, \sigma, b)$ defines a new type inference algorithm that gives a quantitative measure for the λ-calculus in the Linear Rank 2 Quantitative Type System, in the following way:*

1. *<u>If</u> $M = x$, <u>then</u> $\Gamma = [x : \alpha]$, $\sigma = \alpha$ and $b = 0$, where α is a new variable;*
2. *<u>If</u> $M = \lambda x.M_1$ and $\mathsf{T_Q}(M_1) = (\Gamma_1, \sigma_1, b_1)$ <u>then</u>:*
   a. *<u>if</u> $x \notin \mathsf{dom}(\Gamma_1)$, <u>then</u> FAIL;*
   b. *<u>if</u> $(x : \tau) \in \Gamma_1$, <u>then</u> $\mathsf{T_Q}(M) = (\Gamma_{1x}, \tau \multimap \sigma_1, b_1)$;*
   c. *<u>if</u> $(x : \tau_1 \cap \cdots \cap \tau_n) \in \Gamma_1$ (with $n \geq 2$), <u>then</u> $\mathsf{T_Q}(M) = (\Gamma_{1x}, \tau_1 \cap \cdots \cap \tau_n \to \sigma_1, b_1)$.*
3. *<u>If</u> $M = M_1 M_2$, <u>then</u>:*

a. *<u>if</u>* $\mathsf{T}_\mathsf{Q}(M_1) = (\Gamma_1, \alpha_1, b_1)$ *and* $\mathsf{T}_\mathsf{Q}(M_2) = (\Gamma_2, \tau_2, b_2)$,
<u>then</u> $\mathsf{T}_\mathsf{Q}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\alpha_3), b_1 + b_2)$,
where $(\mathbb{S}, \_) = \mathsf{UNIFY}_\mathsf{Q}(\{\alpha_1 = \alpha_2 \multimap \alpha_3, \tau_2 = \alpha_2\})$ *and* α_2, α_3 *are new variables;*

b. *<u>if</u>* $\mathsf{T}_\mathsf{Q}(M_1) = (\Gamma'_1, \tau'_1 \cap \cdots \cap \tau'_n \to \sigma'_1, b_1)$ *(with* $n \geq 2$*) and, for each* $1 \leq i \leq n$*,* $\mathsf{T}_\mathsf{Q}(M_2) = (\Gamma_i, \tau_i, b_i)$,
<u>then</u> $\mathsf{T}_\mathsf{Q}(M) = (\mathbb{S}(\Gamma'_1 + \sum_{i=1}^{n} \Gamma_i), \mathbb{S}(\sigma'_1), b_1 + \sum_{i=1}^{n} b_i + b_3 + 1)$,
where $(\mathbb{S}, b_3) = \mathsf{UNIFY}_\mathsf{Q}(\{\tau_i = \tau'_i \mid 1 \leq i \leq n\})$*;*

c. *<u>if</u>* $\mathsf{T}_\mathsf{Q}(M_1) = (\Gamma_1, \tau \multimap \sigma_1, b_1)$ *and* $\mathsf{T}_\mathsf{Q}(M_2) = (\Gamma_2, \tau_2, b_2)$,
<u>then</u> $\mathsf{T}_\mathsf{Q}(M) = (\mathbb{S}(\Gamma_1 + \Gamma_2), \mathbb{S}(\sigma_1), b_1 + b_2 + b_3 + 1)$,
where $(\mathbb{S}, b_3) = \mathsf{UNIFY}_\mathsf{Q}(\{\tau_2 = \tau\})$*;*

d. *<u>otherwise</u>* FAIL.

Note that b is only increased by 1 and added the quantity given by $\mathsf{UNIFY}_\mathsf{Q}$ in rules 3.(b) and 3.(c), since these are the only cases in which the term M is a redex.

▶ **Example 52.** Let us show the type inference process for the λ-term $\lambda x.xx$.

- By rule 1., $\mathsf{T}_\mathsf{Q}(x) = ([x : \alpha_1], \alpha_1, 0)$.
- By rule 1., again, $\mathsf{T}_\mathsf{Q}(x) = ([x : \alpha_2], \alpha_2, 0)$.
- Then by rule 3.(a), $\mathsf{T}_\mathsf{Q}(xx) = (\mathbb{S}([x : \alpha_1] + [x : \alpha_2]), \mathbb{S}(\alpha_4), 0 + 0) = (\mathbb{S}([x : \alpha_1 \cap \alpha_2]), \mathbb{S}(\alpha_4), 0)$,
where $(\mathbb{S}, \_) = \mathsf{UNIFY}_\mathsf{Q}(\{\alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 = \alpha_3\}) = ([\alpha_3 \multimap \alpha_4/\alpha_1, \alpha_3/\alpha_2], 0)$.
So $\mathsf{T}_\mathsf{Q}(xx) = ([x : (\alpha_3 \multimap \alpha_4) \cap \alpha_3], \alpha_4, 0)$.
- Finally, by rule 2.(c), $\mathsf{T}_\mathsf{Q}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4, 0)$.

▶ **Example 53.** Let us now show the type inference process for the λ-term $(\lambda x.xx)(\lambda y.y)$.

- From the previous example, we have $\mathsf{T}_\mathsf{Q}(\lambda x.xx) = ([\,], (\alpha_3 \multimap \alpha_4) \cap \alpha_3 \to \alpha_4, 0)$.
- By rules 1. and 2.(b), for the identity, the algorithm gives $\mathsf{T}_\mathsf{Q}(\lambda y.y) = ([\,], \alpha_1 \multimap \alpha_1, 0)$.
- By rules 1. and 2.(b), again, for the identity, $\mathsf{T}_\mathsf{Q}(\lambda y.y) = ([\,], \alpha_2 \multimap \alpha_2, 0)$.
- Then by rule 3.(b), $\mathsf{T}_\mathsf{Q}((\lambda x.xx)(\lambda y.y)) = (\mathbb{S}([\,] + [\,] + [\,]), \mathbb{S}(\alpha_4), 0 + 0 + 0 + b_3 + 1) = ([\,], \mathbb{S}(\alpha_4), b_3 + 1)$, where $(\mathbb{S}, b_3) = \mathsf{UNIFY}_\mathsf{Q}(\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\})$, calculated by performing the following transformations:

$$\begin{aligned}\{\alpha_1 \multimap \alpha_1 = \alpha_3 \multimap \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\} &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_1 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\ &\Rightarrow \{\alpha_1 = \alpha_3, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_3\}\\ &\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_2 \multimap \alpha_2 = \alpha_4\}\\ &\Rightarrow \{\alpha_1 = \alpha_4, \alpha_3 = \alpha_4, \alpha_4 = \alpha_2 \multimap \alpha_2\}\\ &\Rightarrow \{\alpha_1 = \alpha_2 \multimap \alpha_2, \alpha_3 = \alpha_2 \multimap \alpha_2, \alpha_4 = \alpha_2 \multimap \alpha_2\}\end{aligned}$$

So $\mathbb{S} = [(\alpha_2 \multimap \alpha_2)/\alpha_1, (\alpha_2 \multimap \alpha_2)/\alpha_3, (\alpha_2 \multimap \alpha_2)/\alpha_4]$
and $b_3 = 1$ because there was performed one transformation (the first) of the form $\{\tau_1 \multimap \tau_2 = \tau_3 \multimap \tau_4\} \cup P \Rightarrow \{\tau_1 = \tau_3, \tau_2 = \tau_4\} \cup P$.
And then, $\mathsf{T}_\mathsf{Q}((\lambda x.xx)(\lambda y.y)) = ([\,], \alpha_2 \multimap \alpha_2, 1 + 1) = ([\,], \alpha_2 \multimap \alpha_2, 2)$.

Since the Quantitative Type Inference Algorithm only differs from the algorithm in Section 3 on the addition of the quantitative measure, and only infers a linear rank 2 intersection type and not a multi-type, the typing soundness (Theorem 54) and completeness (Theorem 55) are formalized in a similar way.

▶ **Theorem 54** (Typing soundness). *If* $\mathsf{T}_\mathsf{Q}(M) = ([x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n], \sigma, b)$*, then* $[x_1 : (\vec{\tau}_1, \vec{t}_1), \ldots, x_n : (\vec{\tau}_n, \vec{t}_n)] \vdash^{b'} M : (\sigma, s)$ *(for some measure* b' *and multi-types* $s, \vec{t}_1, \ldots, \vec{t}_n$*).*

▶ **Theorem 55** (Typing completeness). *If $\Phi \rhd [x_1 : (\vec{\tau}_1, \vec{t}_1), \ldots, x_n : (\vec{\tau}_n, \vec{t}_n)] \vdash^b M : (\sigma, s)$, then $\mathsf{T_Q}(M) = (\Gamma', \sigma', b')$ (for some $\mathbb{T}_{\mathbb{L}1}$-environment Γ', type σ' and measure b') and there is a substitution $\mathbb{S}$ such that $\mathbb{S}(\sigma') = \sigma$ and $\mathbb{S}(\Gamma') \equiv [x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n]$.*

As for the quantitative measure given by the algorithm, we conjecture that it corresponds to the number of evaluation steps of the typed term to normal form, when using the leftmost-outermost evaluation strategy. We strongly believe the conjecture holds, based on the attempted proofs so far and because it holds for every experimental results obtained by our implementation. We have not yet proven this property, which we formalize, in part, in the second point of the strong soundness:

▶ **Conjecture 56** (Strong soundness). *If $\mathsf{T_Q}(M) = ([x_1 : \vec{\tau}_1, \ldots, x_n : \vec{\tau}_n], \sigma, b)$, then:*

1. *There is a derivation $\Phi \rhd [x_1 : (\vec{\tau}_1, \vec{t}_1), \ldots, x_n : (\vec{\tau}_n, \vec{t}_n)] \vdash^{b'} M : (\sigma, s)$ (for some measure b' and multi-types $s, \vec{t}_1, \ldots, \vec{t}_n$);*
2. *If Φ is a tight derivation, then $b = b'$.*

Note that the second point implies, by Theorem 49, that there exists N such that $M \longrightarrow^b N$ and $\mathsf{normal}(N)$, which is what we conjecture.

We believe that proving this conjecture is not a trivial task. A first approach could be to try to use induction on the definition of $\mathsf{T_Q}(M)$. However, this does not work because the subderivations within a tight derivation are not necessarily tight. For that same reason, it is also not trivial to construct a tight derivation from the result given by the algorithm or from a non-tight derivation. Thus, in order to prove this conjecture, we believe that it will be necessary to establish a stronger relation between the algorithm and tight derivations.

5 Conclusions and Future Work

When developing a non-idempotent intersection type system capable of obtaining quantitative information about a λ-term while inferring its type, we realized that the classical notion of rank was not a proper fit for non-idempotent intersection types, and that the ranks could be quantitatively more useful if the base case was changed to types that give more quantitative information in comparison to simple types, which is the case for linear types. We then came up with a new definition of rank for intersection types based on linear types, which we call *linear rank* [23]. Based on the notion of linear rank, we defined a new intersection type system for the λ-calculus, restricted to linear rank 2 non-idempotent intersection types, and a new type inference algorithm which we proved to be sound and complete with respect to the type system.

We then merged that intersection type system with the system for the leftmost-outermost evaluation strategy presented in [1] in order to use the linear rank 2 non-idempotent intersection types to obtain quantitative information about the typed terms, and we proved that the resulting type system gives the correct number of evaluation steps for a kind of derivations. We also extended the type inference algorithm we had defined, in order to also give that measure, and showed that it is sound and complete with respect to the type system for the inferred types, and conjectured that the inferred measures correspond to the ones given by the type system.

In the future, we would like to:

- prove Conjecture 56;
- further explore the relation between our definition of linear rank and the classical definition of rank;
- extend the type systems and the type inference algorithms for the affine terms;
- adapt the Linear Rank 2 Quantitative Type System and the Quantitative Type Inference Algorithm for other evaluation strategies.

References

1 Beniamino Accattoli, Stéphane Graham-Lengrand, and Delia Kesner. Tight typings and split bounds. *Proc. ACM Program. Lang.*, 2(ICFP), July 2018. doi:10.1145/3236789.

2 H. P. Barendregt. Lambda calculi with types. In *Handbook of Logic in Computer Science (Vol. 2): Background: Computational Structures*, pages 117–309. Oxford University Press, Inc., 1993.

3 Hendrik Pieter Barendregt. *The Lambda Calculus - Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1985.

4 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Log. J. IGPL*, 25(4):431–464, 2017.

5 Alonzo Church. A formulation of the simple theory of types. *The Journal of Symbolic Logic*, 5(2):56–68, 1940. URL: http://www.jstor.org/stable/2266170.

6 M. Coppo and M. Dezani-Ciancaglini. An extension of the basic functionality theory for the λ-calculus. *Notre Dame Journal of Formal Logic*, 21(4):685–693, October 1980. doi:10.1305/ndjfl/1093883253.

7 Mario Coppo. An extended polymorphic type system for applicative languages. In Piotr Dembinski, editor, *Mathematical Foundations of Computer Science 1980 (MFCS'80), Proceedings of the 9th Symposium, Rydzyna, Poland, September 1-5, 1980*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer, 1980.

8 H. B. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences*, 20(11):584–590, 1934. doi:10.1073/pnas.20.11.584.

9 Haskell Brooks Curry, Robert Feys, William Craig, J Roger Hindley, and Jonathan P Seldin. *Combinatory Logic*, volume 1. North-Holland Amsterdam, 1958.

10 Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '82, pages 207–212, New York, NY, USA, 1982. Association for Computing Machinery. doi:10.1145/582153.582176.

11 Ferruccio Damiani. Rank 2 intersection for recursive definitions. *Fundamenta Informaticae*, 77(4):451–488, 2007.

12 Daniel de Carvalho. *Sémantiques de la logique linéaire et temps de calcul.* Phd thesis, Université Aix-Marseille II, 2007.

13 Andrej Dudenhefner and Jakob Rehof. Intersection type calculi of bounded dimension. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL '17, pages 653–665, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3009837.3009862.

14 Andrej Dudenhefner and Jakob Rehof. Typability in bounded dimension. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005127.

15 Mário Florido and Luís Damas. Linearization of the lambda-calculus and its relation with intersection type systems. *J. Funct. Program.*, 14(5):519–546, 2004. doi:10.1017/S0956796803004970.

16 Philippa Gardner. Discovering needed reductions using type theory. In *TACS*, volume 789 of *LNCS*, pages 555–574. Springer, 1994.

17 Jean-Yves Girard. Linear logic. *Theor. Comput. Sci.*, 50:1–102, 1987. doi:10.1016/0304-3975(87)90045-4.

18 Trevor Jim. Rank 2 type systems and recursive definitions. *Massachusetts Institute of Technology, Cambridge, MA*, 1995.

19 A. J. Kfoury and J. B. Wells. Principality and decidable type inference for finite-rank intersection types. In *POPL '99, Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–174. ACM, 1999.

20 Assaf Kfoury. A linearization of the lambda-calculus and consequences. *Journal of Logic and Computation*, 10(3):411–436, 2000.

21 Daniel Leivant. Polymorphic type inference. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL)*, pages 88–98, 1983.

22 Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4:258–282, 1982.

23 Fábio Reis, Sandra Alves, and Mário Florido. Linear rank intersection types. In *28th International Conference on Types for Proofs and Programs (TYPES 2022)*, 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_33.pdf.

24 J. A. Robinson. A machine-oriented logic based on the resolution principle. *J. ACM*, 12(1):23–41, January 1965. doi:10.1145/321250.321253.

25 Pawel Urzyczyn. The emptiness problem for intersection types. *The Journal of Symbolic Logic*, 64(3):1195–1215, 1999.

26 Steffen van Bakel. *Intersection type disciplines in lambda calculus and applicative term rewriting systems.* Phd thesis, Mathematisch Centrum, Katholieke Universiteit Nijmegen, 1993.

27 Steffen van Bakel. Rank 2 intersection type assignment in term rewriting systems. *Fundam. Informaticae*, 26(2):141–166, 1996.

A Metatheoretic Analysis of Subtype Universes

Felix Bradley
Royal Holloway, University of London, UK

Zhaohui Luo
Royal Holloway, University of London, UK

Abstract

Subtype universes were initially introduced as an expressive mechanisation of bounded quantification extending a modern type theory. In this paper, we consider a dependent type theory equipped with coercive subtyping and a generalisation of subtype universes. We prove results regarding the metatheoretic properties of subtype universes, such as consistency and strong normalisation. We analyse the causes of undecidability in bounded quantification, and discuss how coherency impacts the metatheoretic properties of theories implementing bounded quantification. We describe the effects of certain choices of subtyping inference rules on the expressiveness of a type theory, and examine various applications in natural language semantics, programming languages, and mathematics formalisation.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Type theory, coercive subtyping, subtype universes

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.9

1 Introduction

Power types were initially introduced by Cardelli as a way of integrating subtyping into a type theory to model bounded quantification [2]. Power(A) represents the collection of subtypes of A, and a given subtyping relation $A \leq B$ can be considered as shorthand for $A : \text{Power}(B)$. Cardelli's system was designed with language design in mind, focusing on behavioural subtyping defined by shared properties of objects. In particular, Cardelli's power types could be used to model a notion of parametric polymorphism called bounded quantification, where one can quantify over the subtypes of a given type. By writing $\lambda(X \leq A).M$ as shorthand for $\lambda(X : \text{Power}(A)).M$.

Cardelli's initial system for power types prioritised expressivity over well-behaved metatheory and included a **Type** : **Type** judgement, which was chosen to express non-terminating computations. Power types have since been revisited by other authors such as Aspinall, who reformulated power types into a predicative system [1]. However, these system have often had issues within the metatheory closely linked to subtyping and bounded quantification. The particular choice of subtyping rules is a common issue, where certain combinations of rules can cause undecidability in the subtyping relation [17, 4].

Maclean and Luo later introduced subtype universes, [16] an analogue of power types designed specifically for extending UTT equipped with coercive subtyping [15, 13]. They showed that this extension preserved metatheoretic properties such as logical consistency and strong normalisation. As subtype universes were initially formulated as an extension of UTT, they are built to work in conjunction with the particular structure of UTT's type universes in mind, which makes for complex proofs. UTT is also restricted in the kind of subtyping rules the system can use, in that subtypes must be present in the same type universe as supertypes, which prevents the use of otherwise useful subtyping rules.

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 9; pp. 9:1–9:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

In this paper, we generalise Maclean and Luo's results by formulating rules for a more expressive notion of subtype universes, designed to extend a more basic dependent type theory. We continue to use coercive subtyping, a method of subtyping best suited for preserving canonicity of terms. Our subtype universes are described by the pseudo-rules

$$\frac{\Gamma \vdash A \,\mathrm{type}}{\Gamma \vdash \mathcal{U}(A)\,\mathrm{type}} \qquad \frac{\Gamma \vdash A \leq_c B}{\Gamma \vdash (A, c) : \mathcal{U}(B)}$$

combined with operators which allow us to retrieve the first object and the second object of the pair. The addition of being able to retrieve the coercion from a subtyping relation allows for subtyping relations using subtypes or bounded quantification. In particular, if the type theory being extended lacks traditional type universes, being able to retrieve coercions allows the type theory to express more complex subtyping relations.

Section 2 describes the implementation of coercive subtyping, outlines the rules used to formulate our generalised notion of subtype universes, discusses what it means for a type theory to lack type universes and why this matters. Section 3 discusses the metatheoretic properties of subtype universes, dependent on the choice of subtyping judgements and rules the underlying type theory is equipped with. In particular, this section analyses an important property of a subset of subtyping judgements: wherein the choice of subtyping relations allows one to "reflect" subtype universes on to the more traditional type universes; and the other case where this is not possible. Section 4 looks at particular choices of subtyping rules and the implications that this work has for the use and application of them. In particular, it focuses on the use of subtyping rules regarding dependent function, universal supertypes, and subtype universes. Finally, section 5 discusses various applications of subtype universes, and showcases several examples of how subtype universes may be used in programming, natural language semantics, and the formalisation of mathematics.

2 Expressive Subtype Universes

In order to be able to introduce subtype universes, we first need to discuss the notion of subtyping and analyse the particular design choice to use coercive subtyping over subsumptive subtyping. From there, we briefly cover Cardelli's power types – designed with programming languages in mind – and Maclean and Luo's prior work on subtype universes – designed for dependent type theories with logic and proofs in mind – and some of the advantages and restrictions of these approaches, before moving on to introducing subtype universes.

2.1 Coercive Subtyping

Introducing subtyping is a very natural extension of any type system, especially when working from a set-theoretic notion or understanding. Subtyping intuitively corresponds to the subset relation, and many properties of subtyping extend from this intuition; for example, we should be able to process any natural number as a rational number, or be able to say that the rational numbers *include* the natural numbers.

When it comes to attempts to implement subtyping, most approaches introduce some form of a new judgement $\Gamma \vdash A \leq B$, read as "the type A is a subtype of the type B", from which we can derive that any term of type A is also a term of type B. This notion as-is without alteration is subsumptive subtyping – any supertype subsumes its subtypes.

This approach runs into issues quickly, however. Subsumptive subtyping as presented breaks the canonicity of a type system: we expect that any object of an inductive type to be computationally equivalent to some canonical object described by the type's rules. With subsumptive subtyping, one can no longer comprehend objects given the computation and elimination rules for the object's type, as that object may actually be of a subtype. As all natural numbers are also rational number, but we can no longer use the rules of rational numbers to process the rational numbers.

One proposed solution to this issue is coercive subtyping [13]. The core concept behind coercive subtyping is that subtyping describes implied coercions that allow us to interpret objects of a subtype as a given canonical form in the supertype. These coercions are functions described by the underlying type theory, allowing us to preserve a lot of the underlying metatheory of the type system. Using the same example as discussed for subsumptive subtyping, we can interpret a natural number as a ration number through the explicit coercion which sends $n \mapsto n/1$.

We can reduce our system with subtyping to a system without subtyping simply by inserting coercions where necessary, and so adding coercive subtyping to a theory tends to be a conservative extension. Of particular use to us is UTT, a modern type theory written in Martin-Löf's Logical Framework, where extending the system with coercive subtyping has been proven to be conservative [15].

▶ Remark 1. We use $\tau[\mathcal{C}]$ to denote both a type theory τ implementing coercive subtyping extended by some set of subtyping judgements (arbitrary or dependent on some other choice), as well as the type theory τ implementing coercive subtyping extended by the specific collection of subtyping judgements $\mathcal{C}$. For example, we later describe a syntactic transformation from τ to UTT[$\mathcal{C}$]: as these systems can't use the same set of subtyping judgements, it can be inferred that the $\mathcal{C}$ in UTT[$\mathcal{C}$] is dependent on the choice of $\mathcal{C}$ in τ.

The key rules for coercive subtyping are as follows:

$$\frac{\begin{array}{c}\Gamma \vdash f : \Pi(x : B).C \\ \Gamma \vdash A <_c B \quad \Gamma \vdash a : A\end{array}}{\Gamma \vdash f(a) : [c(a)/x]C} \qquad \frac{\begin{array}{c}\Gamma \vdash f : \Pi(x : A).C \\ \Gamma \vdash A <_c B \quad \Gamma \vdash a : A\end{array}}{\Gamma \vdash f(a) = f(c(a)) : [c(a)/x]C}$$

These rules (Sub-Intro and Sub-Comp respectively) are used in conjunction with other rules, such as those for congruence, transitivity, and others[1].

Of particular note is the computation rule – in Luo, Soloviev and Xue's own analysis of the metatheory and implementation of coercive subtyping, this is not formally a reduction rule added to the system [15]. Instead, the type theory they describe has a two-step reduction process: the first step is c-reduction, or the insertion of coercions. The second step is the more typical reduction process involving the application of β-reduction. This process is a necessary step due to the need to correctly mark the places in terms where coercions need to be inserted in order for a term to be well-typed before standard reduction can occur.

In this work, we opt to use the same notion of reduction for the type theory we describe – this allows us to treat c-reduction as part of the normal reduction process. In particular, to avoid the complicated metatheory that Luo et al. worked through, we first show that our type theory can be transformed into UTT[$\mathcal{C}$] for particular choices of subtyping judgements. As UTT[$\mathcal{C}$] handles the actual two-step reduction process, this allows us to informally treat c-reduction as on the same level as β-reduction.

When one implements subtyping, one also needs to decide which types are subtypes of which types. This could be both single cases, or families of subtypes (for example, one may wish to say that all finite types are subtypes of $\mathbb{N}$). We consider the most general case possible where the type theory τ is extended by a set of subtyping rules $\mathcal{C}$.

[1] The full set of rules for the implementation of coercive subtyping can be found in [15]. Whilst we do not use the same judgements in this work, the rules are fundamentally the same.

To ensure that τ is sound for a given choice of $\mathcal{C}$, we need a notion of coherence – "that every possible derivation of a statement $\Gamma \vdash a : A$ has the same meaning" [17].We use a similar definition of coherence as Luo et al. as follows:

▶ **Definition 2** (Coherence). *A set of subtyping judgements and inference rules $\mathcal{C}$ is coherent if the following hold:*

- *If* $\Gamma \vdash A <_c B$, *then* $\Gamma \vdash A\,\text{type}$, $\Gamma \vdash B\,\text{type}$, *and* $\Gamma \vdash c : A \to B$
- $\Gamma \nvdash A <_c A$ *for every* Γ, A, *and* c
- *If* $\Gamma \vdash A <_c B$ *and* $\Gamma \vdash A <_{c'} B$, *then* $\Gamma \vdash c = c'$

The coherency of the subtyping judgements and rules used coercive subtyping is critical – without the guarantee of coherency, τ loses any hope of consistency. In practice, reasoning about coherency can be tedious at best. However, other authors have found difficulty within the metatheory of bounded quantification when using subsumptive subtyping [1, 11]. Even for simpler systems, prior authors have provided proofs which were later found to contain errors [17]. Only as recently as 2004 did Compagnoni provide the first proof of the decidability of subtyping for a higher order lambda calculus [6].

2.2 Subtype Universes

Cardelli initially introduced power types as a means of explicitly mechanising bounded quantification – the type Power(A) as the type of subtypes of A [2]. In his original formulation, the judgement $A \leq B$ was in shorthand for A : Power(B), thus typing had completely subsumed subtyping in his system.

Cardelli describes a very expressive type system made possible by these power types, but made compromises in the underlying metatheory of the system in favour of expressiveness. For example, Cardelli's system had a type of all types – while quantification over types is useful, this simple statement can be used to express non-terminating computations, but is also the source of Girard's paradox, which causes logical inconsistency [9, 7, 10]

Maclean and Luo later introduced subtype universes as an extension of UTT[$\mathcal{C}$], UTT equipped with coercive subtyping and a set of subtyping judgements $\mathcal{C}$ [16]. In this implementation, subtyping wasn't completely subsumed by subtype universes; the notions of subtyping and typing were kept disjoint, and subtype universes presented a way for typing to interface with subtyping.

There were some restrictions with Maclean and Luo's presentation, however; they associated subtype universes with the underlying predicative type universes that allowed one to internally quantify over types. This required annotating subtype universes to ensure that types had names in the correct universes, and it also restricted the choice of subtyping relations that could be introduced into the system. In particular, their proof required that for every $A \leq_c B$, A inhabited the same type universe as B.

One of the ways we sought to improve on this design was to expand upon it and remove these restrictions. We use the following rules[2]:

$\mathcal{U}$-Form $$\frac{\Gamma \vdash B\,\text{type}}{\Gamma \vdash \mathcal{U}(B)\,\text{type}}$$

[2] The rules described here only cover types, and do not touch on kinds or subkinding – for the purposes of this work, the rules covering types are sufficient. Whenever we use describe a relation $A \leq_c B$, it is always the case that A and B are types.

$\mathcal{U}$-Intro $$\frac{\Gamma \vdash A \leq_c B}{\Gamma \vdash \langle A, c\rangle : \mathcal{U}(B)}$$

$\mathcal{U}$-σ_1-Elim $$\frac{\Gamma \vdash B\,\text{type} \quad \Gamma \vdash t : \mathcal{U}(B)}{\Gamma \vdash \sigma_1(t)\,\text{type}}$$

$\mathcal{U}$-σ_2-Elim $$\frac{\Gamma \vdash B\,\text{type} \quad \Gamma \vdash t : \mathcal{U}(B)}{\Gamma \vdash \sigma_2(t) : \sigma_1(t) \to B}$$

$\mathcal{U}$-σ_1-Comp $$\frac{\Gamma \vdash B\,\text{type} \quad \Gamma \vdash \langle A, c\rangle : \mathcal{U}(B)}{\Gamma \vdash \sigma_1(\langle A, c\rangle) = B}$$

$\mathcal{U}$-σ_2-Comp $$\frac{\Gamma \vdash B\,\text{type} \quad \Gamma \vdash \langle A, c\rangle : \mathcal{U}(B)}{\Gamma \vdash \sigma_2(\langle A, c\rangle) = c : A \to B}$$

For a given type A, $\mathcal{U}(A)$ is the type of subtypes of A (intuitively, this corresponds to the power set operator). Terms of a subtype universe behave in a similar fashion to pairs, from which we can obtain both the subtype (via the operator σ_1) and the coercion through which we may obtain the corresponding object of the supertype (via the operator σ_2). This design more closely resembles Cardelli's original intent where subtyping is subsumed by typing, as we can now describe any subtyping relation by declaring an object of a subtype universe.

▶ Remark 3. For coherent $\mathcal{C}$, the type of a given $\langle A, c\rangle$ can be calculated by type-checking the term $\sigma_2(\langle A, c\rangle) = c$. One could extend the subtype universes we use here to also explicitly carry information about the supertype, either as part of their data or via annotations. However, assuming that $\mathcal{C}$ is coherent, one is also able to derive the type of any given $\langle A, c\rangle$ by examining the codomain of c. For simplicity, this work does not include these annotations as the metatheory does not fundamentally change.

2.3 Flat Type Theories

This notion that subtyping implies a partial ordering on types in a system is a property we call monotonicity. Under any set-theoretic notion, this seems obvious; the partial ordering would be inclusion. However, if the system has multiple type universes, then monotonicity presents quite a restriction on the choice of subtyping relations one can introduce; there's some natural subtyping relations we may want to use in a system. Consider the example of the type of *pointed subtypes*:

$$\Sigma(x : \mathcal{U}(B)).\sigma_1(x)$$

Intuitively, a pointed subtype of B should also a subtype of B, and so we may want to use the subtyping relation

$$\Sigma(x : \mathcal{U}(B)).\sigma_1(x) \leq_q B$$

where $q \overset{\text{def}}{=} \lambda(y : \Sigma(x : \mathcal{U}(B)).\sigma_1(x)).(\sigma_2(\pi_1(y)))(\pi_2(y))$. However, if we were to follow Maclean and Luo's method for translating $\mathcal{U}(A)$ into an object of UTT[$\mathcal{C}$], we would quickly find that cannot; their method for mapping subtype universes on to type universes simply does not work here. There is a sense in which the left-hand-side of the subtyping relation is more complicated or of a higher order than the right-hand-side – that the LHS should inhabit a higher type universe than the RHS – due to the presence of $\mathcal{U}(B)$.

This leads to one of the key motivations for this work: when can subtype universes be mapped onto type universes? Are there particular choices of subtyping judgements and rules such that the resultant system can't be described by a system with the standard hierarchy of type universes $\mathrm{Type}_1, \mathrm{Type}_2, ...$? Does the choice of subtyping relations affect the metatheory of the system, and if it does, when and how?

In order to better understand how a choice of subtyping relations affects the system, we need to look at a system with coercive subtyping and subtype universes but with minimal structure on its type universes. As we also want to look at the logical consistency of type theories implementing subtype universes, we allow for an impredicative type of propositions Prop.

Whilst the proofs we describe in this work theoretically apply to any "flat" type theory that has no type universes or at most a universe of propositions, we opt to use a subtheory of $\mathrm{UTT}[\mathcal{C}]$ to make several of the proofs in this work more convenient[3] – for example, we use the fact that $\mathrm{UTT}[\mathcal{C}]$ is logically consistent and strongly normalising. In particular, this subtheory of $\mathrm{UTT}[\mathcal{C}]$ contains dependent function types, dependent pair types, an impredicative type universe of propositions, and the atomic types $\mathbf{0}$, $\mathbf{1}$, and $\mathbb{N}$[4].

We write τ to denote the chosen subtheory of $\mathrm{UTT}[\mathcal{C}]$, extended with subtype universes and a (sometimes arbitrary or variable) set of subtyping judgements $\mathcal{C}$. When it is not necessarily clear what specific set of subtyping judgements τ is equipped with, we write $\tau[\mathcal{C}]$ to denote τ equipped with the specific set of subtyping judgements $\mathcal{C}$. We also write $\tau[\mathcal{C}; \mathrm{R}]$ to denote the theory $\tau[\mathcal{C}]$ which has been extended by a specific subtyping judgement or rule R.

3 Metatheory

In our analysis of τ, we will first examine the metatheory of τ equipped with a set of only monotonic subtyping relations. In particular, we will describe an embedding of $\tau[\mathcal{C}]$ in $\mathrm{UTT}[\mathcal{C}]$. In order to describe this embedding, we first need to develop a notion of the level of a type – a measure of it's complexity or "order" under the Curry-Howard interpretation of types as propositions. Afterwards, we will examine the metatheory of the general case where τ is equipped with a set of subtyping relations wherein some are non-monotonic. In both cases, we will prove logical consistency and strong normalisation of τ.

3.1 Type Level

To properly analyse the metatheory of subtype universes, we need to understand under what conditions can subtype universes be reflected on to type universes. In order to do this, we need a notion of the "level" of a type; a description of where a given type fits in the type universe hierarchy. If this notion is well-formed, we will be able to transform terms of a type theory with "well-behaved" subtyping judgements into a type theory where the metatheoretic properties we care about have already been proven.

[3] We use a different set of subtyping judgements to $\mathrm{UTT}[\mathcal{C}]$, such as using $\vdash A\,\mathrm{type}$ rather than $\vdash A : \mathbf{Type}$, but this is primarily for brevity when writing judgements

[4] The atomic types, type constructors, and type of Propositions are all defined using UTT's inductive schemata [12]. While τ can easily be expanded to include inductive data types and inductive propositions, we have elected to skip these inclusions for simplicity and brevity of argument.

In particular, we use UTT[$\mathcal{C}$], Luo's *Unifying Theory of dependent Types* extended with coercive subtyping, as our target theory for this syntactic transformation. This is due to many of the metatheoretic properties we are interested in having been proven for this theory[15, 8]. As such, our own notion of type level is similar in practice to that which Luo uses, but a different approach is necessary; Luo's approach uses type isomorphism and type universes to define level, and we do not have the luxury of the latter [12].

Instead, our notion of type level is defined recursively; basic types (e.g. propositions, Prop, $\mathbb{N} \to \mathbb{N}$, etc.) should be of type level 0, and subtype universes should correspond to increasing the type level by 1.

▶ **Definition 4.** *For a given type A within a context Γ considered in τ, we define its* type level $\mathcal{L}_\Gamma(A)$ *by recursion as follows:*

- *If* $\Gamma \vdash A = P : \mathrm{Prop}$, *then* $\mathcal{L}_\Gamma(A) = 0$;
- *If* $\Gamma \vdash A = \mathrm{Prop}$, $\boldsymbol{0}$, $\boldsymbol{1}$, *or* $\mathbb{N}$, *then* $\mathcal{L}_\Gamma(A) = 0$;
- *If* $\exists B, C$ *such that* $\Gamma \vdash A = \Pi(x : B).C$, *then* $\mathcal{L}_\Gamma(A) = \max_{x:B}\{\mathcal{L}_\Gamma(B), \mathcal{L}_\Gamma(C[x])\}$;
- *If* $\exists B, C$ *such that* $\Gamma \vdash A = \Sigma(x : B).C$, *then* $\mathcal{L}_\Gamma(A) = \max_{x:B}\{\mathcal{L}_\Gamma(B), \mathcal{L}_\Gamma(C[x])\}$;
- *If* $\exists B$ *such that* $\Gamma \vdash A = \mathcal{U}(B)$, *then* $\mathcal{L}_\Gamma(A) = \mathcal{L}_\Gamma(B) + 1$.
- *If* $\exists B, c, s$ *such that* $\Gamma \vdash A = \sigma_1(s)$ *and* $\Gamma \vdash s = \langle B, c\rangle$, *then* $\mathcal{L}_\Gamma(A) = \mathcal{L}_\Gamma(B)$;
- *If* $\exists N$ *such that* $\Gamma, s : \mathcal{U}(N) \vdash A = \sigma_1(s)$ *then* $\mathcal{L}_\Gamma(A) = \max_{M \leq N}\{\mathcal{L}_\Gamma(M)\}$;
- *Otherwise,* $\mathcal{L}_\Gamma(A)$ *is undefined.*

We need to ensure that our notion of type level is well-formed, i.e. every type has a type level, and only types have a type level.

▶ **Lemma 5.** *If* $\Gamma \vdash A\,\mathrm{type}$, *then precisely one of the following hold:*

- $\Gamma \vdash A = P : \mathrm{Prop}$
- $\Gamma \vdash A = \mathrm{Prop}$
- $\Gamma \vdash A = \boldsymbol{0}$
- $\Gamma \vdash A = \boldsymbol{1}$
- $\Gamma \vdash A = \mathbb{N}$
- $\exists B, C$ *such that* $\Gamma \vdash A = \Pi(x : B).C$
- $\exists B, C$ *such that* $\Gamma \vdash A = \Sigma(x : B).C$
- $\Gamma \vdash A = \mathcal{U}(B)$
- $\exists s$ *such that* $\Gamma \vdash A = \sigma_1(s)$

Proof. By induction on derivations of the form $\Gamma \vdash A\,\mathrm{type}$. ◀

▶ **Corollary 6.** $\mathcal{L}_\Gamma(A)$ *is defined if and only if* $\Gamma \vdash A\,\mathrm{type}$.

One of the key metatheoretic features we are interested in is strong normalisation, and so it is also important to check that our notion of type level is invariant under reduction. As discussed in section 2.1, we inherit the same two-step reduction process as that described in Luo, Soloviev and Xue's work on the implementation of coercive subtyping [15]. The first step is c-reduction, wherein coercions are inserted where appropriate to ensure that terms are well-formed and well-typed. The second step is the more typical reduction process via the application of β-reduction. As our goal in this section is to describe an embedding of τ into UTT[$\mathcal{C}$], we can informally treat this reduction process as if it were a single step, putting c-reduction at the same level as β-reduction.

▶ **Definition 7.** *Let $M \rhd N$ denote that applying a single step of reduction to M yields N. Likewise, let $\rhd^*$ denote the reflexive and transitive closure of $\rhd$, i.e. $M \rhd^* N$ denotes that applying 0 or more steps of reduction to M yields N.*

▶ **Theorem 8.** *If $A \rhd B$ then $\mathcal{L}_\Gamma(A) = \mathcal{L}_\Gamma(B)$.*

Proof. To show that this is true in general, we simply need to show that this holds true for reduction via the computation rule $\mathcal{U}$-σ_1-Comp. For any object s of type $\mathcal{U}(B)$, we obtain that it is of the form $\langle B, c\rangle$ for some B and c by induction on derivations. As $\mathcal{L}_\Gamma(\sigma_1(s))$ is defined by its reduction with respect to $\mathcal{U}$-σ_1-Comp, this holds. ◀

▶ **Corollary 9.** *If $A \rhd^* B$ then $\mathcal{L}_\Gamma(A) = \mathcal{L}_\Gamma(B)$.*

▶ Remark 10. Luo's notion of type level had a couple of valuable properties: for example, if two types are type-isomorphic, then they had the same level. For our own notion of type level, this doesn't necessarily hold: if we consider the case of τ with empty $\mathcal{C}$, then every subtype universe is type-isomorphic to the unit type.

There are also other cases which should serve as counterexamples at a glance, but end up being much more interesting under a closer look. Unfortunately, this is outside the scope of this paper.[5]

3.2 A Syntactic Transformation

As previously mentioned, subtype universes can be thought of as a collection of pairs of objects; the first object in the pair is the subtype of the supertype, and the second object of the pair is the coercion through which it is a subtype. We can make this intuition explicit with a syntactic transformation by turning subtype universes in τ into the type of dependent pairs in UTT[$\mathcal{C}$], where the first object is the name of the subtype and the second object is the coercion.

This intuition only works if the subtype *has a name in the corresponding type universe.* This leads us to a formal definition of monotonicity:

▶ **Definition 11.** *A coercive subtyping relation $A \leq_c B$ is* monotonic *if $\mathcal{L}_\Gamma(A) \leq \mathcal{L}_\Gamma(B)$. A set of coercive subtyping judgements and rules $\mathcal{C}$ is* monotonic *if every coercive subtyping relation derived from $\mathcal{C}$ is monotonic.*

However, we've already seen that there are some non-monotonic subtyping judgements which may be desirable, so we will revisit the case of non-monotonic subtyping later; for now, we focus on the case of monotonic subtyping. For monotonic $\mathcal{C}$, we define a syntactic transformation $\delta : \tau \to$ UTT[$\mathcal{C}$] by recursion as described in figure 1.

In this section, we describe both judgements derived in τ and judgements derived in UTT[$\mathcal{C}$]. While one can view the underlying type theory T as a subtheory of UTT, we use a different set of judgements. As such, we distinguish between them where necessary; any judgement derived in τ will be denoted with $\vdash_\tau$, and any judgement derived in UTT[$\mathcal{C}$] will be denoted with $\vdash_{\mathrm{UTT}}$. Moreover, any context in τ will be written as Γ, and any context in UTT[$\delta(\mathcal{C})$] will be written as $\delta(\Gamma)$

To ensure this is a useful transformation, we need to check whether or not types have names in the expected type universes; whether we can derive the judgements we expect regarding translated terms; and whether this transformation preserves metatheoretic properties we're interested in, such as logical consistency and strong normalisation.

Prior to this, however, we need to discuss the translation of subtyping and how δ can preserve any notion of subtyping. For some derivation $\Gamma \vdash_\tau A \leq_c B$, we expect to be able to take any $\Gamma \vdash a : A$ and derive that $\Gamma \vdash a : B$; if δ is to preserve subtyping, then we also need to ensure that not only can we derive $\delta(\Gamma) \vdash_{\mathrm{UTT}} \delta(a) : \delta(A)$, but also that $\delta(\Gamma) \vdash_{\mathrm{UTT}} \delta(a) : \delta(B)$.

[5] We encourage the curious reader to think about the following example: for $\Gamma, a : A \vdash P : \mathrm{Prop}$, consider whether or not the types $\Pi(a : A).P$ and $\Pi(X : \mathcal{U}(A)).\Pi(x : \sigma_1(X)).P[c(x)/a]$ are type-isomorphic.

$$\delta(\mathcal{U}(B)) = \Sigma(X : \mathrm{Type}_{\mathcal{L}_\Gamma(B)}).(X \to \delta(B)) \qquad \delta(\sigma_2) = \pi_2$$

$$\delta(\langle A, c\rangle) = (\mathbf{t}^{\mathcal{L}_\Gamma(B)}_{\mathcal{L}_\Gamma(A)} \circ \mathbf{n}(\delta(A)), \delta(c)) \qquad \delta(\sigma_1) = \mathbf{T} \circ \pi_1$$

$$\delta(\Pi_{x:A}B) = \Pi_{x:\delta(A)}\delta(B) \qquad \delta(\lambda(a : A).B) = \lambda(a : \delta(A)).\delta(B)$$

$$\delta(\Sigma_{x:A}B) = \Sigma_{x:\delta(A)}\delta(B) \qquad \delta((a, b)) = (\delta(a), \delta(b))$$

$$\delta(\pi_1) = \pi_1 \qquad \delta(\pi_2) = \pi_2 \qquad \delta(f(x)) = \delta(f)(\delta(x))$$

$$\delta(\mathbf{0}) = \mathbf{0} \qquad \delta(\mathbf{1}) = \mathbf{1} \qquad \delta(*) = *$$

$$\delta(\mathbb{N}) = \mathbb{N} \qquad \delta(0) = 0 \qquad \delta(\mathrm{S}) = \mathrm{S}$$

$$\delta(\mathrm{Prop}) = \mathrm{Prop} \qquad \delta(\forall(x : A).P) = \mathbf{Prf}(\forall(x : \delta(A)).\delta(P))$$

$$\delta(\Lambda(a : A).P) = \Lambda(a : \delta(A)).\delta(P)$$

Figure 1 The transformation of terms in $\tau[\mathcal{C}]$ into terms in $\mathrm{UTT}[\delta(\mathcal{C})]$ under δ.

As we have only defined the domain of our transformation δ as being $\mathrm{UTT}[\mathcal{C}]$ for *some* $\mathcal{C}$, we can exactly specify our target theory by choosing which subtyping judgements and rules it uses, depending on our initial choice of $\mathcal{C}$ for τ. As such, we extend our definition of δ to include the subtyping judgements of τ, sending any $A \leq_c B \in \mathcal{C}$ to $\delta(A) \leq_{\delta(c)} \delta(B)$; we write the collection of the latter as $\delta(\mathcal{C})$. Moreover, we can precisely say that, for $\mathcal{C}$ a fixed set of subtyping judgements and rules, we can define a syntactic transformation $\delta : \tau \to \mathrm{UTT}[\delta(\mathcal{C})]$ per the above.

▶ **Theorem 12.** *If* $\Gamma \vdash A\,\mathrm{type}$*, then* $\delta(\Gamma) \vdash \delta(A) : \mathbf{Type}$ *and there exists some* $i \in \omega$ *and some term* n *in* $\mathrm{UTT}[\delta(\mathcal{C})]$ *such* $\delta(\Gamma) \vdash n : \mathrm{Type}_i$ *and* $\mathbf{T}_i(n) = \delta(A)$*.*

Proof. Proof by induction on derivations of $\Gamma \vdash A\,\mathrm{type}$ and cross-referencing with lemma 5. We consider the following cases:

Case 1. $\exists B, P$ such that A is of the form $\forall(b : B).P$. As in UTT the predicate $\forall(x : A).P$ exists for any type A and any predicate P over A and has a name in Prop, we may take $i = 0$ and n to be the name of $\mathbf{Prf}(\forall(b : \delta(B)).\delta(P))$ in Type_0.

Case 2. A of the form Prop. Trivially, we may take $i = 0$ and $n = \mathit{prop} : \mathrm{Type}_0$.

Case 3. A of the form $\mathbf{0}$, $\mathbf{1}$, or $\mathbb{N}$. We take advantage of UTT's rules which introduce the names of inductive data types to establish that, as all of these constructors do not have any types in their generating sequence of inductive schemata, $i = 0$ and $\delta(A)$ must have names in Type_0 [12].

Case 4. $\exists B, C$ such that A is of the form $\Pi(x : B).C$, or $\Sigma(x : B).C$. Similarly, these types have a name in Type_i if and only if both $\delta(B)$ has a name in Type_j and $\delta(C)$ has a name in Type_k. Assuming $\delta(B)$ has a name in Type_j and $\delta(C)$ has a name in Type_k, we may take $i = \max\{j, k\}$ and thus $\delta(A)$ has a name in Type_i, which is as desired.

Case 5. $\exists B$ such that A is of the form $\mathcal{U}(B)$. Using the same reasoning as per Π types: as $\delta(A) = \Sigma(X : \mathrm{Type}_{\mathcal{L}_\Gamma(B)}).(X \to \delta(B))$, we note that $\mathrm{Type}_{\mathcal{L}_\Gamma(B)}$ has a name in $\mathrm{Type}_{\mathcal{L}_\Gamma(B)+1}$ and that $X \to \delta(B)$ has a name in $\mathrm{Type}_{\mathcal{L}_\Gamma(B)}$, and so we may take n as the name for $\mathrm{Type}_{\mathcal{L}_\Gamma(B)}$.

Case 6. $\exists s$ such that A is of the form $\sigma_1(s)$. By induction, we have some B and c such that $\Gamma \vdash s = \langle B, c\rangle$ and thus $A = B$ by $\mathcal{U}$-σ_1-Comp. We may take $i = \mathcal{L}_\Gamma(A)$ and n to be the name of $\delta(B)$ in $\mathrm{Type}_{\mathcal{L}_\Gamma(A)}$.

Case 7. $\exists B$ such that A is of the form $\sigma_1(s)$, where $s : \mathcal{U}(B)$ is variable. As $\mathcal{C}$ is monotonic, we know that $\mathcal{L}_\Gamma(A)$ is at most $\mathcal{L}_\Gamma(B)$ and thus we may take $i = \mathcal{L}_\Gamma(B)$ and n to be the name of $\delta(A)$ in $\text{Type}_{\mathcal{L}_\Gamma(B)}$. ◀

▶ **Theorem 13.** *For coherent and monotonic $\mathcal{C}$, the rules of $\tau[\mathcal{C}]$ are admissible in* $\text{UTT}[\delta(\mathcal{C})]$ *under transformation by δ.*

Proof. This is a special case of theorem 23 taking $k = 0$. ◀

▶ **Lemma 14.** *Let R be a coherent subtyping judgement or rule. Then $\delta(R)$ is coherent.*

Proof. Assume that $R \vdash A \leq_c B$. By our definition of δ, we immediately have that $\delta(\Gamma) \vdash \delta(c) : \delta(A) \to \delta(B)$. Injectivity of δ implies both: that $\delta(\Gamma) \not\vdash \delta(A) \leq_{\delta(c)} \delta(A)$ for every Γ, A, and c; and that if $\delta(\Gamma) \vdash \delta(A) \leq_{\delta(c)} \delta(B)$ and $\delta(\Gamma) \vdash \delta(A) \leq_{\delta(c')} \delta(B)$, then $\delta(\Gamma) \vdash c = c'$. Thus $\delta(A) \leq_{\delta(c} \delta(B)$ is coherent for every derivation of $A \leq_c B$ from R. ◀

3.3 On Monotonic Subtyping

▶ **Theorem 15** (Logical consistency). *For monotonic $\mathcal{C}$, τ is logically consistent, i.e. there does not exist some Γ and p such that $\Gamma \vdash p : \forall(P : \text{Prop}).P$.*

Proof. Proof by contradiction. Assume that there does exist some Γ and p such that $\Gamma \vdash p : \forall(P : \text{Prop}).P$. Under syntactic transformation by δ, we obtain $\delta(\Gamma) \vdash \delta(p) : \mathbf{Prf}(\forall(P : \text{Prop}).P)$, which contradicts the logical consistency of $\text{UTT}[\mathcal{C}]$. ◀

▶ **Theorem 16** (Preservation of one-step reduction). *For monotonic $\mathcal{C}$, if $M \rhd N$ then $\delta(M) \rhd \delta(N)$.*

Proof. Proof by induction on the terms of τ. For every reduction $M \rhd^R N$ in τ via a computation rule R, we show that there exists a computation rule S such that $\delta(M) \rhd^S \delta(N)$ in $\text{UTT}[\mathcal{C}]$.

As before, there are several trivial cases which have been omitted for brevity, most of which are special cases of the computation rule μ for UTT's inductive types[6]. We focus on the non-trivial cases regarding subtyping and subtype universes.

Case 1. $f(a) \rhd^{\text{Sub-Comp}} f(c(a)) \Rightarrow$

$$\delta(f(a)) \overset{\text{def}}{=} \delta(f)(\delta(a)) \rhd^{\text{CA}_2} \delta(f)(\delta(c)(\delta(a))) \overset{\text{def}}{=} \delta(f)(\delta(c(a))) \overset{\text{def}}{=} \delta(f(c(a)))$$

Case 2. $\sigma_1(\langle A, c\rangle) \rhd^{\mathcal{U}\text{-}\sigma_1\text{-Comp}} A \Rightarrow$

$$\delta(\sigma_1(\langle A, c\rangle)) \overset{\text{def}}{=} \delta(\sigma_1)(\delta(\langle A, c\rangle)) \overset{\text{def}}{=} \mathbf{T}_{\mathcal{L}_\Gamma(B)} \circ \pi_1(\mathbf{t}^{\mathcal{L}_\Gamma(B)}_{\mathcal{L}_\Gamma(A)} \circ \mathbf{n}(\delta(A)), \delta(c))$$
$$\rhd^{\Sigma_1} \mathbf{T}_{\mathcal{L}_\Gamma(B)}(\mathbf{t}^{\mathcal{L}_\Gamma(B)}_{\mathcal{L}_\Gamma(A)} \circ \mathbf{n}(\delta(A))) \overset{\text{def}}{=} \delta(A)$$

Case 3. $\sigma_2(\langle A, c\rangle) \rhd^{\mathcal{U}\text{-}\sigma_2\text{-Comp}} c \Rightarrow$

$$\delta(\sigma_2(\langle A, c\rangle)) \overset{\text{def}}{=} \delta(\sigma_2)(\delta(\langle A, c\rangle)) \overset{\text{def}}{=} \pi_2((\mathbf{t}^{\mathcal{L}_\Gamma(B)}_{\mathcal{L}_\Gamma(A)} \circ \mathbf{n}(\delta(A)), \delta(c))) \rhd^{\Sigma_2} \delta(c)$$ ◀

▶ **Corollary 17** (Preservation of multi-step reduction). *For monotonic $\mathcal{C}$, if $M \rhd^* N$ then $\delta(M) \rhd^* \delta(N)$.*

▶ **Theorem 18** (Strong normalisation). *For monotonic $\mathcal{C}$, if $\Gamma \vdash M : A$, then M is strongly normalisable, i.e. every possible sequence of reductions of M is finite.*

[6] These include Π types, Σ types, $\mathbb{N}$, $\mathbf{1}$, $\mathbf{0}$.

Proof. Proof by contradiction. Assume that there does exist some Γ and M such that $\Gamma \vdash M : A$ where M has an infinite reduction sequence. Under transformation by δ we obtain $\delta(M)$. As δ preserves multi-step reduction, we obtain an infinite reduction sequence of $\delta(M)$, which contradicts the strong normalisation of UTT[$\mathcal{C}$]. ◀

▶ Remark 19. For monotonic $\mathcal{C}$, we can take advantage of our transformation δ and the decidability of type-checking in UTT[$\mathcal{C}$] to show that in τ both type checking and the subtyping relation is decidable. For any given term M in τ, we can consider the type of $\delta(M)$. As UTT[$\mathcal{C}$] is a conservative extension[7] of UTT, we can type-check $\delta(M)$. As δ is injective, we are also able to know the form of the type of $\delta(M)$ in UTT[$\mathcal{C}$] and thus also in τ.

To show that subtyping is also decidable, for any given pair of types A, B, we can consider the construction of a term t which is well-typed if and only if the subtyping relation $A \leq B$ is derivable (such as $\lambda(f : B \to \mathbb{N}).\lambda(a : A).f(a)$.) [17]. By checking if $\delta(t)$ is well-typed in UTT[$\mathcal{C}$], we can decide whether or not $A \leq B$.

3.4 On Non-Mononotic Subtyping

When analysing the more general case of the metatheory of $\tau[\mathcal{C}]$, where the set of subtyping judgements $\mathcal{C}$ contains some non-monotonic subtyping relations, one will often run into immediate difficulty. Our first approach to this problem was to try and use the notion that the use of coercive subtyping is a kind of shorthand for the insertion of exact coercions. The extension of a type theory with coercive subtyping should be a conservative extension, and likewise extending a type theory with additional subtyping rules should not affect the underlying theory [15, 13].

One could interpret the extension of a type theory with additional coercive subtyping rules as a kind of weakly conservative extension – it should not "add" new types to the theory, and one should not suddenly be able to obtain terms in types that were previously uninhabited. Furthermore, if you have a term that depends on the existence of a subtyping judgement, then it should be possible to construct another term of the same type that *doesn't* rely on that subtyping judgement through the insertion of coercions – this can form the basis of a type-checking algorithm, assuming you have a type-checking algorithm for the case where $\mathcal{C}$ is empty.

When applying this idea to τ, the existence of subtype universes causes immediate problems. If you have a term that depends on an object of a subtype universe, say $\langle A, c\rangle : \mathcal{U}(B)$, then you can attempt to construct another object of the same type by both inserting coercions and by replacing instances of $\langle A, c\rangle$ with $\langle B, \mathrm{id}_B\rangle$. At the term-level, this idea requires effort, but is sound. Unfortunately, the idea does not work in general due to the presence of new types.

Consider a subtyping rule from which we may derive $A \leq_c B$, and extending $\tau[\mathcal{C}]$ with R. Observe that

$$\Sigma(x : \mathcal{U}(B)).\, \mathrm{Eq}_{\mathcal{U}(B)}(x, \langle A, c\rangle)$$

where $\mathrm{Eq}_A \overset{\text{def}}{=} \forall(x : A).\forall(y : A).\forall(P : \mathrm{Prop}).(P(x) \leftrightarrow P(y))$ is the type of propositional Leibniz equality on a given type A. Whilst this type can be derived in $\tau[\mathcal{C}; \mathrm{R}]$, it cannot be derived in $\tau[\mathcal{C}]$, and there's no obvious process from which we may try to extend the type-checking algorithm for $\tau[\mathcal{C}]$.

[7] More accurately, UTT[$\mathcal{C}$] is equivalent to a type theory which is a conservative extension of UTT – the exact meaning of "conservativity" of τ with respect to T is not always clear with the rules for coercive subtyping we have used.

David Aspinall's work on the predicative typed lambda calculus λ_{Power} lead to him introducing a notion of "rough types" [1]. Where as one may intuit our first approach described above as an attempt to blur terms together to extend a type-checking algorithm, Aspinall's approach instead blurs types together, organises them into rough types, and develops a rough-type-checking algorithm. Aspinall shows that, as long as there is a method to calculate the rough type of a given term, this is sufficient to be able to prove strong normalisation for the calculus.

We believe Aspinall's approach would also work for τ. However, it also possible to further generalise the definitions and proofs given in sections 3.2 and 3.3 to cover practically most non-monotonic subtyping relations. The key insight is that most non-monotonic subtyping relations are still relatively well-behaved: by measuring how far a subtyping relation is from being monotonic, we're able to adjust the embedding of τ into UTT[$\mathcal{C}$] in response. We first introduce this measurement:

▶ **Definition 20.** *A coercive subtyping relation $A \leq_c B$ is* k-monotonic *if* $\mathcal{L}_\Gamma(B) - \mathcal{L}_\Gamma(B) + k \geq 0$. *A set of coercive subtyping judgements and rules $\mathcal{C}$ is* k-monotonic *if every coercive subtyping relation derived from $\mathcal{C}$ is k-monotonic.*

This is a generalisation of monotonicity of subtyping – in particular, any given monotonic subtyping relation is 0-monotonic. If we consider the example of pointed subtypes introduced earlier in section 2.3

$$\Sigma(x : \mathcal{U}(B)).\sigma_1(x) \leq_q B,$$

we can calculate that the difference between the level of the supertype and the level of the subtype is independent of the choice of the type B.

$$\begin{aligned}\mathcal{L}_\Gamma(B) - \mathcal{L}_\Gamma(\Sigma(x : \mathcal{U}(B)).\sigma_1(x)) &= \mathcal{L}_\Gamma(B) - \max(\mathcal{L}_\Gamma(B) + 1, \mathcal{L}_\Gamma(\sigma_1(x))) &(1)\\ &= \min(-1, -\mathcal{L}_\Gamma(\sigma_1(x))) &(2)\end{aligned}$$

If you wanted to extend $\tau[]$ with the rule $B\,\text{type} \vdash \Sigma(x : \mathcal{U}(B)).\sigma_1(x) \leq_q B$, then this subtyping rule would be 1-monotonic. At worst, because a type can only be constructed with a finite number of $\mathcal{U}$s, we know that there must always exist some k such that this subtyping rule is k-monotonic.

▶ **Corollary 21.** *Let R be an i-monotonic coherent coercive subtyping rule, and let $\mathcal{C}$ be a j-monotonic set of coherent coercive subtyping judgements. Then $[\mathcal{C}; R]$ is (at worst) $(i + j)$-monotonic.*

From here, one can modify the embedding δ described in section 3.2: if you have a bounded measure k of the difference in level between a supertype and a subtype, then this says that if you were attempting to embed $\tau[\mathcal{C}]$ into UTT[$\mathcal{C}$] by mapping $\mathcal{U}(B)$ to $\Sigma(X : \text{Type}_{\mathcal{L}_\Gamma(B)}).(X \to \delta(B))$, then the particular choice of $\text{Type}_{\mathcal{L}_\Gamma(B)}$ is k levels under where it needs to be for $\delta(A)$ to have a name.

▶ **Definition 22.** *For a given set of k-monotonic subtyping judgements $\mathcal{C}$, define a syntactic transformation $\delta_k : \tau[\mathcal{C}] \to \text{UTT}[\delta_k(\mathcal{C})]$ defined identically to δ except*

$$\delta_k(\mathcal{U}(B)) \overset{def}{=} \Sigma(X : \text{Type}_{\mathcal{L}_\Gamma(B)+k}).(X \to \delta_k(B)), \quad \delta_k(\langle A, c\rangle) = (\mathbf{t}_{\mathcal{L}_\Gamma(A)}^{\mathcal{L}_\Gamma(B)+k} \circ \mathbf{n}(\delta_k(A)), \delta_k(c))$$

As there is only a minor difference between δ_k and δ, it's easy to see that a lot of the proofs needed to show that δ_k is a well-behaved embedding that preserves term reduction are almost identical to the proofs for δ, except for the extra terms of k, as seen in the proof of theorem 23. As a result, the proofs of theorem 12 and lemma 14 are functionally identical, as are the proofs regarding logical consistency, term reduction and strong normalisation.

▶ **Theorem 23.** *For coherent and k-monotonic $\mathcal{C}$, the rules of $\tau[\mathcal{C}]$ are admissible in* $\mathrm{UTT}[\delta(\mathcal{C})]$ *under transformation by δ_k.*

Proof. As τ is a subtheory of $\mathrm{UTT}[\mathcal{C}]$ as discussed in section 2.3, the majority of the rules of τ are effectively derivable in $\mathrm{UTT}[\mathcal{C}]$ by default. We omit the trivial cases (such as rules for the unit type, dependent function types, etc.) and instead focus on the non-trivial cases regarding coercive subtyping and subtype universes.

$$\delta_k\text{-Sub-Intro}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(f):\Pi(x:\delta_k(B)).\delta_k(C)\\ \delta_k(\Gamma)\vdash\delta_k(A)<_{\delta_k(c)}\delta_k(B)\quad \delta_k(\Gamma)\vdash\delta_k(a):\delta_k(A)\end{array}}{\delta_k(\Gamma)\vdash\delta_k(f)(\delta_k(a)):[\delta_k(c)(\delta_k(a))/x]\delta(C)}\quad \textit{derivable}$$

$$\delta_k\text{-Sub-Comp}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(f):\Pi_{x:\delta_k(A)}\delta_k(C)\\ \delta_k(\Gamma)\vdash\delta_k(A)<_{\delta_k(c)}\delta_k(B)\quad \delta_k(\Gamma)\vdash\delta_k(a):\delta_k(A)\end{array}}{\Gamma\vdash\delta_k(f)(\delta_k(a))=\delta_k(f)(\delta_k(c(a))):[\delta_k(c)(\delta_k(a))/x]\delta_k(C)}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-Form}\quad \frac{\delta_k(\Gamma)\vdash\delta_k(B)\,\mathrm{type}}{\delta_k(\Gamma)\vdash\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B)):\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k+1}}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-Intro}\quad \frac{\delta_k(\Gamma)\vdash\delta_k(A)<_{\delta_k(c)}\delta_k(B)}{\delta_k(\Gamma)\vdash\delta_k(\langle A,c\rangle):\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B))}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-}\sigma_1\text{-Elim}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(B):\mathrm{Type}_{\mathcal{L}_\Gamma(B)}\\ \Gamma\vdash\delta_k(t):\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B))\end{array}}{\Gamma\vdash\mathbf{T}_{\mathcal{L}_\Gamma(B)+k}\circ\pi_1(\delta_k(t)):\mathbf{Type}}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-}\sigma_2\text{-Elim}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(B):\mathrm{Type}_{\mathcal{L}_\Gamma(B)}\\ \delta_k(\Gamma)\vdash\delta_k(t):\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B))\end{array}}{\delta_k(\Gamma)\vdash\pi_2(\delta_k(t)):\pi_1(\delta_k(t))\to\delta_k(B)}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-}\sigma_1\text{-Comp}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(B):\mathrm{Type}_{\mathcal{L}_\Gamma(B)}\\ \delta_k(\Gamma)\vdash(\delta_k(A),\delta_k(c)):\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B))\end{array}}{\delta_k(\Gamma)\vdash\mathbf{T}_{\mathcal{L}_\Gamma(B)+k}\circ\pi_1(\delta_k(\langle A,c\rangle))=\delta_k(A):\mathbf{Type}}\quad \textit{derivable}$$

$$\delta_k\text{-}\mathcal{U}\text{-}\sigma_2\text{-Comp}\quad \frac{\begin{array}{c}\delta_k(\Gamma)\vdash\delta_k(B):\mathrm{Type}_{\mathcal{L}_\Gamma(B)}\\ \delta_k(\Gamma)\vdash\delta_k(A),\delta_k(c)):\Sigma(X:\mathrm{Type}_{\mathcal{L}_\Gamma(B)+k}).(X\to\delta_k(B))\end{array}}{\delta_k(\Gamma)\vdash\pi_1(\delta_k(\langle A,c\rangle))=\delta_k(c):\delta_k(A)\to\delta_k(B)}\quad \textit{derivable}$$

◀

▶ **Theorem 24** (Logical consistency)**.** *For k-monotonic $\mathcal{C}$, τ is logically consistent, i.e. there does not exist some Γ and p such that $\Gamma\vdash p:\forall(P:\mathrm{Prop}).P$.*

▶ **Theorem 25** (Preservation of one-step reduction)**.** *For k-monotonic $\mathcal{C}$, if $M\rhd N$ then $\delta(M)\rhd\delta(N)$.*

▶ **Corollary 26** (Preservation of multi-step reduction)**.** *For k-monotonic $\mathcal{C}$, if $M\rhd^* N$ then $\delta(M)\rhd^*\delta(N)$.*

▶ **Theorem 27** (Strong normalisation)**.** *For k-monotonic $\mathcal{C}$, if $\Gamma \vdash M : A$, then M is strongly normalisable, i.e. every possible sequence of reductions of M is finite.*

▶ Remark 28. As in the case with monotonic subtyping, as the embedding δ_k is injective, we can type-check any given term M of $\tau[\mathcal{C}]$ with k-monotonic $\mathcal{C}$ by type-checking the term $\delta_k(M)$ in $\mathrm{UTT}[\delta(\mathcal{C})]$.

4 On Subtyping and Bounded Quantification

Works on subtyping and on specific type systems or programming languages with an implementation of subtyping or bounded quantification often have a variety of basic subtyping rules or judgements used to enrich the type system. Some of these are particularly popular amongst authors due to their power, or their ability in making for an expressive type system. When it comes to the metatheory of subtyping, particular instances or combinations of subtyping rules can also often cause problems with regards to normalisation and logical consistency, but also in often desirable properties, such as the decidability of subtyping.

4.1 With Dependent Functions

Bounded quantification was first introduced by Cardelli and Wegner in the language *Fun*, with a handful of subsumptive subtyping rules to introduce non-trivial subtypes into the system [3]. *Fun* has been a core for study and analysis, and several variations, simplifications and extensions have come about. In a paper analysing the subtyping of one of these variations called *minimal bounded Fun*, Pierce proves that the subtyping relation is undecidable by encoding the halting problem as a subtyping problem [17].

In particular, the interaction between two subtyping rules causes this undecidability; the existence of a universal supertype **Top**, and a dependent function subtyping rule.

$$\frac{\Gamma \vdash A\,\mathrm{type}}{\Gamma \vdash A \leq \mathbf{Top}} \qquad \frac{\Gamma \vdash B_1 \leq A_1 \quad \Gamma, \alpha \leq B_1 \vdash A_2 \leq B_2}{\Gamma \vdash \forall(\alpha \leq A_1).A_2 \leq \forall(\alpha \leq B_1).B_2}$$

Top alone can cause a plethora of issues (as discussed later in this section), but the function subtyping rule used is of particular interest. Castagna and Pierce have spoken about the issues this rule presents, and have discussed several variations [4]. However, the version presented above can also be presented in coercive subtyping.

▶ **Lemma 29** (Coherency of Π-Infer)**.** *The subtyping rule*

$$\Pi\text{-}\mathit{Infer} \qquad \frac{\begin{array}{c}\Gamma \vdash A_1 \leq_c B_1 \quad \Gamma, a : A_1 \vdash A_2\,\mathrm{type} \\ \Gamma, b : B_1 \vdash B_2\,\mathrm{type} \quad \Gamma, a : A_1 \vdash [c(a)/b]B_2 \leq_d A_2\end{array}}{\Gamma \vdash \Pi(b : B_1).B_2 \leq_q \Pi(a : A_1).A_2}$$

is coherent, where

$$q = \lambda(g : \Pi(b : B_1).B_2).\lambda(a : A_1).d(a, g(c(a)))$$

Proof. Per the definition of coherency, we have three conditions we need to check. First, we check that q has the expected type.

$$q : (\Pi(x : B_1).B_2) \to (\Pi(x : A_1).A_2)$$

Secondly, we check the case where $A_1 = B_1$ and $B_2 = A_2$, and thus c is the identity function id_{A_1} and d is the constant function $\lambda(x : A_1).\, \mathrm{id}_{A_2}$. We obtain, through various computation rules,

$$\begin{aligned} q &= \lambda(g : \Pi(x : A_1).A_2).\lambda(x : A_1).d(x, g(c(x))) && (3)\\ &= \lambda(g : \Pi(x : A_1).A_2).\lambda(x : A_1).(\lambda(x : A_1).\, \mathrm{id})(x, g(\mathrm{id}(x))) && (4)\\ &= \lambda(g : \Pi(x : A_1).A_2).\lambda(x : A_1).\, \mathrm{id}(g(\mathrm{id}(x))) && (5)\\ &= \lambda(g : \Pi(x : A_1).A_2).\lambda(x : A_1).g(x) && (6)\\ &= \mathrm{id}_{\Pi(x:A_1).A_2)} && (7) \end{aligned}$$

as desired. Similarly, for the third case, it's easy to see that coherency of the hypothesis implies equality for multiple different derivations of q. ◀

One of the primary hurdles with this subtyping rule is that how the context is filled with regards to the codomains is uncertain, which may be evidence of an issue. Splitting this rule into two, with one for contravariance in the domain and the other for covariance in the codomain, is also possible [13] and arguably easier: Cardelli and Wegner's original formulation of the language *Fun* only uses a subtyping rule for the codomain, which can be shown to be decidable [17].

4.2 With Universal Supertypes

In coercive subtyping, the implementation of a universal supertype is often impossible for the sheer reason that it cannot be implemented coherently – **Top** cannot contain a single object[8], and so needs to be able to describe every possible object of the system. For extremely simple type theories, such as a theory containing only finite types and no type constructors, this is relatively trivial; but for even marginally more complex theories, the complexity and size of **Top** grows rapidly.

Even ignoring coherency issues for a moment, the same approach we have taken with respect to the metatheory of reflecting subtype universes on to type universes can't be taken. Adding a universal supertype **Top** to τ always results in non-monotonic subtyping; if we choose any $n \in \omega$ such that $\mathcal{L}_\Gamma(\mathbf{Top}) \overset{\text{def}}{=} n$, we can always find a subtype of strictly greater level (such as $\mathcal{U}^{n+1}(\mathbf{1})$.)

Introducing bounded quantification in conjunction with **Top** into a system also has an immediate concern in the semantics of the type $\forall(X \leq \mathbf{Top}).X$. Equivalently, in a system where we have mechanised bounded quantification via subtype universes or power types, we can consider the type $\mathcal{U}(\mathbf{Top})$ – for all intents and purposes, this should be a type of all types. By Girard's paradox, these systems should be non-normalising and thus inconsistent[9] [7].

Under the set-theoretic containment semantics, any universal supertype has to be transfinite in nature in the same way that a type of all types is transfinite in nature. It may be possible to "solve" the metatheoretic issues that universal supertypes present by taking a similar approach to type universes: by replacing **Top** with a series of partial supertypes $\mathbf{Top}_1, \mathbf{Top}_2, ...$ equipped with subtyping relations $\mathbf{Top}_1 \leq \mathbf{Top}_2 \leq ....$

[8] Assuming the system has at least two distinct terms!

[9] There are several interesting routes through which one may attempt to obtain a proof of this inconsistency; the traditional approach here is to obtain what is essentially a bijection between a type and its power type, which is a contradiction by the diagonal lemma [10, 19]. Another possible route may be that, through subtype universes, a type theory may be capable of modelling itself – how large this model may be is unclear, however.

In fact, with subtype universes, such a set of partial supertypes can completely replace the typical use of type universes by replacing quantification over Type_i with bounded quantification over $\mathbf{Top}_i$. Furthermore, i doesn't necessarily need to be indexed by ω; one can take any partially ordered set I and, for $i, j \in I$, let $\mathbf{Top}_i \leq \mathbf{Top}_j$ whenever $i \leq j$.

4.3 With Subtype Universes

A subtyping inference rule for power types introduced by both Cardelli and Aspinall is as follows:

$$\frac{\Gamma \vdash A \leq B}{\Gamma \vdash \mathrm{Power}(A) \leq \mathrm{Power}(B)}$$

We can form an equivalent rule for τ as follows:

$$\frac{\Gamma \vdash A \leq_c B}{\Gamma \vdash \mathcal{U}(A) \leq_{\lambda(X:\mathcal{U}(A)).\langle \sigma_1(X), c \circ \sigma_2(X) \rangle} \mathcal{U}(B)}$$

which is well-typed and thus coherent by transitivity of subtyping. Under the set-theoretic notion of subtypes as subsets, this is also an extremely useful rule; we can reason about collections of subsets. On the other hand, this also greatly impacts any higher structure on subtype universes; we may wish to reason about the subtypes of $\mathcal{U}(B)$ without taking into account the subtypes of B itself.

There is also an issue of whether subtype universes and subtyping should be allowed to interact in the first place; subtype universes are an extension of a system with subtyping, and one may consider that system to have already had a set of subtyping relations judgements and rules implemented. Even then, being able to reason about subtyping relations with subtype universes can still be useful. We consider the following example:

$$O(n) \overset{\text{def}}{=} \Sigma(x : \mathbb{N}).(x < n)$$

$$O(n) \leq_{\pi_1} \mathbb{N}$$

$$\mathbb{N} \leq_{\lambda(n:\mathbb{N}).\langle O(n), \pi_1 \rangle} \mathcal{U}(\mathbb{N})$$

This example of ordinals-as-types was derived from looking at the logical consistency of certain subtyping relations and attempting to recreate Girard's paradox [7]. However, the two above subtyping judgements have an interesting property; the coherency of the second subtyping judgement now depends on the former. By allowing subtyping judgements to quantify over subtype universes, the coherency of any one subtyping judgement becomes dependent on the other judgements in the system.

4.4 Decidability of Typing and Subtyping

While we have sketched a proof that type-checking is decidable for monotonic subtyping and a subset of non-monotonic subtyping (i.e. those which are k-monotonic), there is still the open question of whether or not non-monotonic subtyping is decidable in general (i.e. for $\mathcal{C}$ that are non-monotonic but where there does not exist a k such that $\mathcal{C}$ is k-monotonic). Furthermore, our results rely primarily on the advantages that coercive subtyping brings: whether or not these ideas apply to systems that use subsumptive subtyping is left unanswered, especially for non-monotonicity.

There do exist examples of non-monotonic subtyping being decidable, such as Compagnoni's proof for System $F^{\omega}_{\wedge}$ [6]. $F^{\omega}_{\wedge}$ uses subsumptive subtyping and the dependent function subtyping rule, but drops the universal supertype **Top** in favour of empty intersection types quantifying over a kind. Additionally, Aspinall's work on power types lead to the development of rough-typing [1] – a kind of approximate type-checking that's powerful enough to still prove results such as strong normalisation. The algorithm Aspinall outlines provides enough information that one could likely refine it into a full type-checking algorithm.

In both Aspinall's work on power types and in Hutchins' work on pure subtype systems, [11] the authors point out that bounded quantification can be used to subsume a notion of kinds. Aspinall emulates the Edinburgh Logical Framework in λ_{Power}, and his rough type-checking can be seen as a notion of kinding . Likewise, Hutchins describes a process through which functions using kinds can be equivalently described through bounded quantification over types without any loss of generality.

This does open up several questions, such as whether you can "retrofit" kinds into a pure type system with bounded quantification. The difficulty of a generation or inversion lemma when working without coercive subtyping lends to one fear in regards to the combined use of both **Top** and power types. **Top** $\leq$ **Top** allows one to express non-terminating computations with power types; in a system where the distinction between terms and types are blurred, is it possible to form non-normalising types, just as one can express non-normalising terms?

5 Applications

5.1 Bounded Quantification

The mechanisation of bounded quantification was one of the key motivations for introducing power types in Cardelli's original paper [2]. Cardelli described a focus on the expressiveness of his system at the cost of non-terminating type-checking, but his formulation considered the case where subtyping was entire subsumed by typing (i.e. $A \leq B$ as shorthand for $A : \text{Power}(B)$). Maclean and Luo's subtype universes showed that the mechanisation of bounded quantification could preserve metatheoretic properties, but also kept typing and subtyping disjoint enough that subtype universes could lead to a more expressive system [16].

Subtype universes as described in this paper are capable of modelling bounded quantification; one should consider $\lambda(A \leq B).M$ as shorthand for $\lambda(x : \mathcal{U}(B)).[\sigma_1(x)/A]M$. This is particularly useful when it comes to record types. For example, consider a function

$$\text{darken} : \{\text{luminosity} : \text{Float32}\} \to \text{Float32} \to \{\text{luminosity} : \text{Float32}\}$$

This function is clearly sufficient in the case where we're handling objects that only carry luminosity data, but if we were to use subtyping to parse an object which also carried hue and saturation data, then we would receive an object with only luminosity data back. To fix this issue, we can use bounded quantification, and instead use the function

$$\text{darken} : \Pi(X : \mathcal{U}(\{\text{luminosity} : \text{Float32}\})).\sigma_1(X) \to \text{Float32} \to \sigma_1(X)$$

which allows us to preserve any excess information parsed in.

Subtyping can also be taken into consideration and used when designing languages and software to prevent errors. Often a collection of types designed to model information or objects will have some higher notion of structure on them; for example, the type $\mathbb{Q}$ equipped with addition, subtraction, multiplication and division forms a field. When designing the data types used to model these objects, we may wish for these operations to be as close to independent of which type we're considering them in. For example, take Int16 $\leq_c$ Float32. For any two x, y : Int16, we would expect $c(x+y) = c(x) + c(y)$, and we may wish to choose a c or change our definitions of $+$ accordingly.

These approaches make for future-safe design and development of software. Often during the development of software one may wish to refactor code to improve its maintainability, reduce complexity, or prepare for adding new features; by taking these safe-guarding measures in the design-process, errors can be prevented and type-safety can be ensured. We also retain one of the key advantages of subtype universes and power types in that these objects can be interpreted as types; we can consider functions that range over types, which is not possible with just bounded quantification. Our system is also capable of modelling new kinds of subtyping relations through this process.

For example, for a given type B, consider the type of pointed subtypes $\Pi(x : \mathcal{U}(B)).\sigma_1(x)$. Intuitively, a pointed subtype of B is also a subtype of B, but Maclean and Luo's subtype universes had no way of describing this subtyping relation coherently. However, with the introduction of σ_2 in our system, we can obtain the exact coercion through which one type is a subtype of another type, and so we can use the coherent subtype relation

$$\Sigma(x : \mathcal{U}(B)).\sigma_1(x) \leq_q B \text{ where } q \overset{\text{def}}{=} \lambda(y : \Pi(x : \mathcal{U}(B)).\sigma_1(x)).(\sigma_2(\pi_1(y)))(\pi_2(y)).$$

With Maclean's subtype universes, we could implement behavioural subtyping with relative ease, but one could not describe subtyping relations which used bounded quantification. Being able to combine the two makes for a more expressive system.

5.2 Natural Langauge Semantics

Subtyping has a variety of applications in natural language semantics in describing the relationships between different categories and groups. When we start formalising these relationships, there quickly becomes a desire for some notion of bounded quantification.

Montague grammar, introduced in Richard Montague's seminal work, solves this problem by interpreting categorisation as propositions [18]. By semantically typing different language constructs, we can interpret a fully constructed sentence as a type. For example, we could interpret the sentence "all grass is green" as a term of type $\forall x.\text{isGrass}(x) \to \text{isGreen}(x)$. As lemongrass is a type of grass and thus $\text{isLemongrass}(x) \leq$ isGrass(x), we would also obtain the sentence "all lemongrass is green"[10].

However, we quickly run into an issue with Montague grammar in that we can form nonsense sentences: we can semantically type the sentence "all purple is trains" or "the month of December plays football", but these sentences don't make sense and are likely undesirable. We can instead model natural language semantics in a modern type theory, where every category of objects has its own type and subtyping is used to describe the relationships between categories of objects [14, 5]. For example, we can consider the type of Woman as a subtype of Human, or Chair as a subtype of Furniture. This allows us to use subtype universes to model categorisation of objects. Using $\mathcal{U}$-Infer as an example, one may infer from Fish $\leq$ Animals that $\mathcal{U}(\text{Fish}) \leq \mathcal{U}(\text{Animals})$ – i.e. that a type of species of fish is also type of species of animal.

We can also use subtype universes to model subsective adjectives. For example, how should one interpret the adjective "skillful" versus the adjective "small"? Let CN be the universe of common nouns. For any common noun, the interpretation of small : $\Pi(A$:

[10] While one may understand types as propositions via the Curry-Howard correspondence, the subtleties of subtyping with propositions in a type theory where propositions are treated distinctly from types is still an unexplored topic. Further analysis and discussion on this is outside the scope of this work, however.

CN).$A \to$ Prop is both sound and meaningful. However, using the same idea to obtain skillful : $\Pi(A : \text{CN}).A \to$ Prop presents some issues. Whilst skillful(Doctor) makes sense, an example such as skillful(Chair) is obviously not intended. If we wish to exclude unintended combinations from our modelling of language, we can instead consider the semantic typing skillful : $\Pi(A : \mathcal{U}(\text{Human})).\sigma_1(A) \to$ Prop. Of course, as Doctor $\leq_c$ Human, we have that skillful($\langle$Doctor$, c\rangle$) is a well-typed expression. However, this now excludes unintended cases – skillful($\langle$Chair$, c'\rangle$) is ill-typed because Chair $\not\leq$ Human.

5.3 Point-Set Topology

Subtyping has some interesting relationships with topology. For example, one could choose a set of subtyping judgements and rules such that a type and its subtypes model a space and its open sets. Under this application, the subtype universe of a space corresponds to its topology – the set of open sets.

This is a relatively easy process if the space we want to look at has a given metric, as the topology derived from a metric space is given by the union of open balls around points. As an example, we consider the rational numbers with the Euclidean metric (denoted d). We first model the rational numbers $\mathbb{Q}$ as $\mathbb{N} \times \mathbb{N}/0$ with addition, multiplication, metric, and ordering defined in the typical ways, e.g. we define the Euclidean metric $d : \mathbb{Q} \to \mathbb{Q} \to \mathbb{Q}$ such that $d(x, y) = |x - y|$.

We then consider the following three coherent subtyping rules:

$$\frac{\Gamma \vdash z : \mathbb{Q}, \quad \Gamma \vdash r : \mathbb{Q}}{\Gamma \vdash \Sigma(x : \mathbb{Q}).(d(z,x) < r) \leq_{\pi_1} \mathbb{Q}} \qquad \frac{\Gamma \vdash I \text{ type} \quad \Gamma, x : I \vdash A \leq_c \mathbb{Q}}{\Gamma \vdash \Sigma(i : I)A \leq_{\lambda(p:\Sigma(i:I)A).c(\pi_1(p),\pi_2(p))} \mathbb{Q}}$$

$$\frac{\Gamma \vdash A <_c \mathbb{Q} \quad \Gamma \vdash B <_{c'} \mathbb{Q}}{\Gamma \vdash \Sigma(a : A).\Sigma(b : B).(c(a) = c'(b)) <_{c \circ \pi_1} \mathbb{Q}}$$

These three rules are sufficient for $\mathcal{U}(\mathbb{Q})$ to be a topology of $\mathbb{Q}$. It's rather simple to check that there exists an empty subtype; that the arbitrary union of subtypes is also a subtype; and that the intersection of two subtypes is also a subtype.

We do, however, have a multitude of technical and semantic issues to work our way through. Is this the correct notion of union and intersection, for example? Have we chosen our basis correctly, or is there a different basis for the topology which is more convenient to work with (for example, slicing the real line)?

Under the above rules, there exists multiple empty subtypes – whilst the rules we've introduced could be further refined to prevent these issues, we may also wish to reason about $\mathcal{U}(\mathbb{Q})$ as a setoid. Similarly, we may also want to reason about $\mathbb{Q}$ as a setoid as there exists multiple different ways of expressing the same rational number: for example, $1/2$ is represented by the pairs $(1, 2)$, $(2, 4)$, $(3, 6)$, and so on. There is an obvious notion of propositional equality $\text{Eq}_{\mathbb{Q}}$ we can equip to $\mathbb{Q}$ by defining

$$\text{Eq}_{\mathbb{Q}}(p, q) \overset{\text{def}}{=} (\pi_1(p - q) = 0) : \text{Prop}\,.$$

However, with a notion of point-set topology formalised, we can also equip $\mathbb{Q}$ with a notion of equality based on open sets.

$$\text{Eq}_{\mathbb{Q}}{}'(p, q) \overset{\text{def}}{=} \forall(x : \mathcal{U}(\mathbb{Q})).((\exists(r : \sigma_1(x)).(p = r)) \wedge (\exists(s : \sigma_1(x)).(q = s))).$$

Whilst $\text{Eq}_{\mathbb{Q}}$ is certainly more reasonable for reasoning about arithmetic or number theory, it's plausible that $\text{Eq}_{\mathbb{Q}}{}'$ and similar notions may be more useful for reasoning about continuity, limits, or Cauchy sequences. Exploring this further is outside the scope of this paper, however.

When using subtype universes to model topologies, the coercions used in the subtyping judgements can be understood as mapping open sets to open sets. We leave open the question of whether these coercions can be interpreted as a continuous embedding of one space into another and what this means for the semantics of a type theory. Nonetheless, understanding subtyping as continuous embedding could provide some new intuition for the problems regarding universal supertypes: **Top** is not only a universal supertype, but also a space in which every space in a type theory can be embedded into. If we want to be able to use a universal super type with coercive subtyping, then we need some way to describe every object of our type theory.

We can immediately ask questions and draw conclusions about what such a space looks like: for example, if we take a type theory consisting only of finite types with no type constructors, then **Top** is described by $\mathbb{N}$. For any type theory modelling anything more complicated, **Top** is unlikely to look like a slice of $\mathbb{R}^\infty$, as any space embedding into $\mathbb{R}^\infty$ must be both seperable and metrizable[11].

6 Conclusion

This work generalised and extended Maclean and Luo's prior notion of subtype universes in order to provide support for a much wider range of coercive subtyping relations. By examining a type system lacking the traditional type universe hierarchy of $\mathrm{Type}_0, \mathrm{Type}_1, \mathrm{Type}_2, ...$, we have allowed for subtypes more complicated than their supertype; and by allowing one to obtain the coercion through the σ_2 operator, we are able to express coherent subtyping judgements and rules that use subtype universes.

In doing so, we have found that the metatheory remains relatively well-behaved regardless of the choice in subtyping relations; both monotonic and k-monotonic subtyping result in logical consistency and strong normalisation of terms. Additionally, we have sketched a proof of the decidability of type-checking and subtyping in both cases by embedding terms into $\mathrm{UTT}[\mathcal{C}]$ and type-checking there instead. However, whether or not similar results can be proven for non-monotonic subtyping in general (i.e. where there does not exist a k such that subtyping is k-monotonic) is left open.

Throughout the course of this paper, we hope to have shed some light regarding particular uses of subtyping, such as the difficulties of using a universal supertype **Top**. However, we have left several questions open, such as whether or not τ is confluent, the subtleties of subtyping between propositions, and how closely linked subtyping and subtype universes are to a notion of continuous embeddings.

In particular, we want to more closely examine Hutchins' work on pure subtype systems [11]: systems akin to pure type systems wherein the typing relation is entirely subsumed by the subtyping relation, a notion "almost completely dual to [...] the approach taken by Cardelli" . Simple questions such as the decidability of subtyping or how one may implement a notion of propositional logic into a pure subtype system are left open, and we look forward to working on this in the future.

References

1 David Aspinall. Subtyping with power types. In Peter G. Clote and Helmut Schwichtenberg, editors, *Computer Science Logic*, pages 156–171, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[11] More formally, every seperable and metrizable space is homeomorphic to a subset of the Hilbert cube $[0,1]^\infty$, which is a subspace of $\mathbb{R}^\infty$. This is established in the proof of Urysohn's metrization theorem.

2 Luca Cardelli. Structural subtyping and the notion of power type. In *Proceedings of the 15th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '88, pages 70–79, New York, NY, USA, 1988. Association for Computing Machinery. `doi:10.1145/73560.73566`.

3 Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, December 1985. `doi:10.1145/6041.6042`.

4 Giuseppe Castagna and Benjamin C. Pierce. Decidable bounded quantification. In *Proceedings of the 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '94, pages 151–162, New York, NY, USA, 1994. Association for Computing Machinery. `doi:10.1145/174675.177844`.

5 Stergios Chatzikyriakidis and Zhaohui Luo. On the interpretation of common nouns: Types versus predicates. In Stergios Chatzikyriakidis and Zhaohui Luo, editors, *Modern Perspectives in Type-Theoretical Semantics*, pages 43–70. Springer Cham, 2017. `doi:10.1007/978-3-319-50422-3`.

6 Adriana Compagnoni. Higher-order subtyping and its decidability. *Information and Computation*, 191(1):41–103, 2004. `doi:10.1016/j.ic.2004.01.001`.

7 Thierry Coquand. An analysis of Girard's paradox. Technical Report RR-0531, INRIA, May 1986. URL: `https://hal.inria.fr/inria-00076023`.

8 Healfdene Goguen. *A Typed Operational Semantics for Type Theory*. PhD thesis, University of Edinburgh, 1994.

9 Douglas J. Howe. The computational behaviour of girard's paradox. Technical report, Cornell University, Ithaca, New York, USA, March 1987.

10 Antonius J. C. Hurkens. A simplification of Girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Typed Lambda Calculi and Applications*, pages 266–278, Berlin, Heidelberg, 1995. Springer Berlin Heidelberg.

11 DeLesley S. Hutchins. Pure subtype systems. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 287–298, New York, NY, USA, 2010. Association for Computing Machinery. `doi:10.1145/1706299.1706334`.

12 Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, London, March 1994.

13 Zhaohui Luo. Coercive subtyping. *Journal of Logic and Computation*, 9(1):105–130, February 1999. `doi:10.1093/logcom/9.1.105`.

14 Zhaohui Luo. Common nouns as types. In D. Béchet and A. Dikovsky, editors, *Proceedings of the 7th International Conference on Logical Aspects of Computational Linguistics (LACL'12)*, pages 173–185, Berlin, Heidelberg, 2012. Springer-Verlag.

15 Zhaohui Luo, Sergey Soloviev, and Tao Xue. Coercive subtyping: Theory and implementation. *Information and Computation*, 223:18–42, 2013. `doi:10.1016/j.ic.2012.10.020`.

16 Harry Maclean and Zhaohui Luo. Subtype Universes. In Ugo de'Liguoro, Stefano Berardi, and Thorsten Altenkirch, editors, *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, volume 188 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:16, Dagstuhl, Germany, 2021. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. `doi:10.4230/LIPIcs.TYPES.2020.9`.

17 Benjamin C. Pierce. Bounded quantification is undecidable. In *Proceedings of the 19th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '92, pages 305–315, New York, NY, USA, 1992. Association for Computing Machinery. `doi:10.1145/143165.143228`.

18 Richmond H. Thomason, editor. *Formal Philosophy: Selected Papers of Richard Montague*. Yale University Press, New Haven, 1974.

19 Kevin Watkins. Hurkens' simplification of Girard's paradox, July 2004. URL: `https://www.cs.cmu.edu/~kw/research/hurkens95tlca.elf`.

The Münchhausen Method in Type Theory

Thorsten Altenkirch
School of Computer Science, University of Nottingham, UK

Ambrus Kaposi
Eötvös Loránd University, Budapest, Hungary

Artjoms Šinkarovs
Heriot-Watt University, Edinburgh, Scotland, UK

Tamás Végh
Eötvös Loránd University, Budapest, Hungary

Abstract

In one of his long tales, after falling into a swamp, Baron Münchhausen salvaged himself and the horse by lifting them both up by his hair. Inspired by this, the paper presents a technique to justify very dependent types. Such types reference the term that they classify, e.g. $x : F\ x$. While in most type theories this is not allowed, we propose a technique on salvaging the meaning of both the term and the type. The proposed technique does not refer to preterms or typing relations and works in a completely algebraic setting, e.g categories with families. With a series of examples we demonstrate our technique. We use Agda to demonstrate that our examples are implementable within a proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Type structures

Keywords and phrases type theory, proof assistants, very dependent types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.10

Supplementary Material *Software*: `https://bitbucket.org/akaposi/combinator/src/master/post-types2022`, archived at `swh:1:dir:9609c613a09bd552cbca4546d247e3cefebef9cd`

Funding *Ambrus Kaposi*: Supported by the "Application Domain Specific Highly Reliable IT Solutions" project which has been implemented with support from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme, and by Bolyai Scholarship BO/00659/19/3.
Artjoms Šinkarovs: Supported by the Engineering and Physical Sciences Research Council through the grant EP/N028201/1.

1 Introduction

When we want to understand how powerful the given type system is, we identify objects that the given type is allowed to depend on. For instance, in simply-typed systems types are built from a fixed set of ground types and operations. In System F we introduce type variables and binders making it possible to define new operations that compute types. In dependently-typed systems we are allowed to compute types from terms.

At the same time, we rarely explore dependencies within a typing relation. For example, consider the case when the type is allowed to depend on the term that it is typing:

$$x : F\,x$$

Such a situation is often referred to as *very dependent type* [10]. The immediate two questions arise: (i) does this ever occur in practice? (ii) how do you support this within a type system?

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 10; pp. 10:1–10:20
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Let us consider an example where such a type occurs very naturally. There is a well-known type isomorphism, saying that pairs can be represented as functions from boolean:

$$A \times B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\; b \;\mathsf{then}\; A \;\mathsf{else}\; B.$$

Consider now upgrading the isomorphism to dependent product on the left hand side. Given $A : \mathsf{Type}$, $B : A \to \mathsf{Type}$, we want something like

$$\Sigma\, A\, B \;\cong\; (b : \mathsf{Bool}) \to \mathsf{if}\; b \;\mathsf{then}\; A \;\mathsf{else}\; (B\, \square),$$

but what do you put in the placeholder $\square$? It should be the output of the function when the input is $b = \mathsf{true}$. Once the function is given a name, we can refer to it:

$$f : (b : \mathsf{Bool}) \to \mathsf{if}\; b \;\mathsf{then}\; A \;\mathsf{else}\; (B\, (f\, \mathsf{true}))$$

Supporting such definitions in a type system can be tricky. Hickey gives [10] a type system with very dependent functions using pre-terms and typing relations [5]. However, it turns out that many very dependent types can be understood algebraically and even encoded in proof assistants.

Many practical examples are easier to understand when very dependent types are present. One familiar example is the use of type universes in proof assistants such as Coq or Agda. Both systems use Russell universes, and if we ignore the universe levels, Set is of type Set in Agda, and Type is of type Type in Coq. This is clearly the case of very dependent types.

Our main observation is that algebraic presentation requires cutting the cycle of a given very dependent type. This is achieved by introducing a temporary placeholder type and a number of equations that eliminate the placeholders. The proposed scheme can be summarised as follows. For a very dependent type $(x : \mathsf{F}\; x)$ find:

$$\mathsf{A} : \mathsf{Set};\; \mathsf{G} : \mathsf{A} \to \mathsf{Set};\; \alpha : \{a : \mathsf{A}\} \to \mathsf{G}\; a \to \mathsf{A}$$

Such that F can be decomposed in $\mathsf{G} \circ \alpha$. In this case, a very dependent type can be expressed as the following triplet:

```
a  : A       - Placeholder
x  : G a     - The data
eq : a ≡ α x - Closing the cycle
```

This approach works if these equations are propositional, but it forces a lot of transport along the newly introduced equations (this situation is commonly referred to as transport hell). In Agda we can turn these propositional equations into definitional ones by means of rewrite rules or forward declarations.

The main contribution of this paper lies in applying the Münchhausen method to five practical examples. Our setting is Martin-Löf type theory extended with function extensionality, UIP (uniqueness of identity proofs) and forward declarations. From [12] we know that such a formulation without forward declarations is conservative with respect to its intensional version. This means that, in principle, all the presented types that do not use forward declarations can be given in intensional type theory, but in a much more verbose way. We conjecture that the same holds for the type theory with forward declarations as well, but as these are not very well understood, we would not claim this.

We use Agda to demonstrate concrete implementation of our examples, but there is nothing Agda-specific in the method itself. In Agda, the Münchausen method can be realised in four different ways:

1. Identity types and explicit equations (Section 3);
2. Forward declarations (Sections 3, 4 and 5);
3. Shallow embedding as described in [14] (Sections 6 and 6);
4. Postulates and rewrite rules (Section 7)

While it is not yet clear whether all very dependent types as defined in [10] can be handled by the proposed method, we believe that the examples that we provide give a first step towards answering this question.

This paper is an Agda script, therefore all the examples in the paper have been typechecked.

The content of this paper was presented at the TYPES'22 conference in Nantes [3].

2 Background

In this section we give a brief introduction to Agda, which is an implementation of Martin-Löf's dependent type theory [16] extended with a number of constructions such as inductive data types, records, modules, *etc.* We make a brief overview of the features that are used in this paper. For the in-depth introduction please refer to Agda's user manual [1].

2.1 Datatypes

Datatypes are defined as follows:

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data Fin : ℕ → Set where
  zero : {n : ℕ} → Fin (suc n)
  suc  : {n : ℕ} → Fin n → Fin (suc n)
```

The type ℕ of unary natural numbers is a datatype with two constructors: zero and suc. The type of ℕ is Set which is Agda's builtin type of small types.

The type Fin is indexed by ℕ and it also has two constructors zero and suc. The names of the constructors can overlap. In the definition of the Fin constructors we used implicit argument syntax[1] to define the variable n. When using constructors of Fin, we can leave out specifying these arguments relying on Agda's automatic inference. These can be also passed explicitly as follows:

```
a : Fin 2
a = zero {n = 1}
```

Numbers 0, 1, 2, . . . are implicitly mapped into ℕ in the usual way.

2.2 Records

Agda makes it possible to define records[2]. They generalise dependent products, making it possible to name the fields. For example, we can define the type of dependent pairs using records as follows:

[1] https://agda.readthedocs.io/en/v2.6.3/language/implicit-arguments.html

[2] https://agda.readthedocs.io/en/v2.6.3/language/record-types.html

```
record Pair (A : Set) (B : A → Set) : Set where
  constructor _,_
  field
    fst : A
    snd : B fst
```

The Pair record is parametrised by the type A and the family B (over A). The type has two fields named fst and snd that correspond to first and second projections of the dependent pair. Finally, we can give a constructor _,_ that we can use to construct the values of type Pair. Note that the constructor uses the mixfix notation[3]. This means that arguments replace the underscores, so the comma , becomes a binary operation. The values can be constructed as follows:

```
b : Pair ℕ Fin
b = 5 , zero
```

2.3 Modules

Modules[4] make it possible to collect the definitions that logically belong together, giving them a separate namespace. Modules can accept parameters. They abstract variables for the definitions within the module. In the paper we only use modules to group the definitions together and reuse the names of the definitions. For example, here we define modules X and Y, where Y is parametrised with the variable n, which is a natural number.

```
module X where
  foo : ℕ
  foo = 5

module Y (n : ℕ) where
  foo : ℕ
  foo = n
```

2.4 Forward Declarations

Agda makes it possible[5] to make a declaration and provide a definition later. This is useful when dealing with mutual definitions. For example, we can have a mutual definition of even and odd numbers as the following indexed types:

```
data Even : ℕ → Set
data Odd : ℕ → Set

data Even where
  zero : Even zero
  suc : {n : ℕ} → Odd n → Even (suc n)
data Odd where
  suc : {n : ℕ} → Even n → Odd (suc n)
```

[3] https://agda.readthedocs.io/en/v2.6.3/language/mixfix-operators.html

[4] https://agda.readthedocs.io/en/v2.6.3/language/module-system.html

[5] https://agda.readthedocs.io/en/v2.6.3/language/mutual-recursion.html#mutual-recursion-forward-declaration

First we defined the signature of both data types; after that we gave the definitions of their constructors. By making such a forward declaration, we were able to refer Odd in the definition of the suc constructor in Even.

2.5 Postulates

Agda makes it possible[6] to declare objects without ever providing a definition. This can be thought of as a typed free variable. For example, we can postulate that there exists some natural number q:

```
postulate
  q : ℕ
```

2.6 Rewrite Rules

Agda makes it possible to define rewrite rules[7], which are typically used to turn propositional equations into definitional ones. However, in combination with postulates, we can also simulate some reduction behaviour. For example, we can postulate natural numbers, eliminator for natural numbers and reduction equalities. Then we can use rewrite rules to simulate reduction.

```
postulate
  Nat : Set
  z : Nat
  s : Nat → Nat
  elim : (P : Nat → Set) → P z → ((n : Nat) → P n → P (s n))
       → (n : Nat) → P n
  elim-z : ∀ {P pz ps} → elim P pz ps z ≡ pz
  elim-s : ∀ {P pz ps n} → elim P pz ps (s n) ≡ ps n (elim P pz ps n)
  {-# REWRITE elim-z elim-s #-}
```

We postulate the type for natural numbers Nat and its two constructors z and s. After that, we postulate the type for the eliminator for natural numbers in the usual way. Finally we define two β-like equalities for the eliminator. By turning these equalities into rewrite rules, we make our eliminator to reduce in the usual way.

3 Dependent Sequences

We start with a detailed exploration of the dependent product isomorphism presented in the introduction. While this example is not very practical, it is concise and easy to understand.

For a fixed pair of types, the encoding of non-dependent pair can be expressed in Agda as follows:

```
pair : (b : Bool) → if b then String else ℕ
pair true  = "Types" - first projection
pair false = 22      - second projection
```

[6] https://agda.readthedocs.io/en/v2.6.3/language/postulates.html
[7] https://agda.readthedocs.io/en/v2.6.3/language/rewriting.html

Surprisingly, similar presentation of dependent pairs for a fixed type and a family over it is expressible in Agda using forward declarations. For example, for ℕ and Fin we have:

```
dpair-hlpr : ℕ
dpair : (b : Bool) → if b then ℕ else Fin dpair-hlpr
dpair-hlpr = dpair true

dpair true  = 5   - first projection of type ℕ
dpair false = # 3 - second projection of type Fin 5
```

According to Münchhausen method, we “cut” the cyclic dependency of *dpair* by introducing a placeholder called *dpair-hlpr*. Here forward declarations make it possible to postpone the definition of *dpair-hlpr*. After that, we define *dpair* and we “close the cycle” by giving the value to the placeholder.

Let us try to abstract this encoding to arbitrary types, and prove the isomorphism from the introduction. For non-dependent pairs, we have:

```
module _ (ext : ∀ {a b} → Extensionality a b) where
  Pair : Set → Set → Set
  Pair A B = (b : Bool) → if b then A else B

  Pair≅× : ∀ A B → (A × B) ↔ Pair A B
  Pair≅× A B = mk↔ {f = to}{from} (to∘from , λ _ → refl)
    where
      to : _; from : _; to∘from : _
      to (a , b) = λ {true → a; false → b}
      from    f = f true , f false
      to∘from f = ext λ {true → refl; false → refl}
```

The ↔ is a type for bijections, and mk↔ constructs the bijection from forward and backward functions and a pair of proofs that they are inverses of each other. As can be seen, conversion from Pair is memoisation. Correspondingly, conversion into Pair is “unmemoisation”. These operations are clearly inverses of each other, assuming functional extensionality.

Encoding of dependent pairs has to mention the placeholder h and the value that this placeholder gets (f true) by means of explicit equation eq.

```
record DPair (A : Set) (B : A → Set) : Set where
  constructor _▷_[_]
  field
    h  : A
    f  : (b : Bool) → if b then A else B h
    eq : h ≡ f true
```

Such an encoding corresponds to the first variant of the Münchhausen method, as the equality that closes the cycle is made explicit. Note that eq corresponds to the definition of *dpair-hlpr* in the presentation above.

The isomorphism between dependent pairs and DPair requires a little bit more work, as we are dealing with equations within the structure. Assuming functional extensionality ext and uniqueness of identity proofs uip, equality of two DPairs can be derived from point-wise pair equality given by $\_\equiv^{d}\_$.

```
module _ (ext : ∀ {a b} → Extensionality a b)
         (uip : ∀ {A : Set}{a b : A} → (p q : a ≡ b) → p ≡ q) where

  record _≡ᵈ_ {A}{B} (a b : DPair A B) : Set where
    constructor _&_
    field
      fst  : DPair.h a ≡ DPair.h b
      snd : DPair.f a false ≡ subst B (sym fst) (DPair.f b false)

  ≡ᵈ⇒≡ : ∀ {A B} {a b : DPair A B} → a ≡ᵈ b → a ≡ b
```

With these definitions at hand, the isomorphism between DPairs and Σ types is very similar to its non-dependent version:

```
  Pair≅Σ : ∀ A B → (Σ A B) ↔ DPair A B
  Pair≅Σ A B = mk↔ {f = to}{from} (to∘from , λ _ → refl)
    where
      to (a , b) = a ▹ (λ {true → a; false → b}) [ refl ]
      from (h ▹ f [ eq ]) = f true , subst B eq (f false)
      to∘from (h ▹ f [ eq ]) = ≡ᵈ⇒≡ (sym eq & cong (λ x → subst B x (f false)) (sym∘sym eq))
```

As can be seen, working with explicit equalities is tricky. Switching to more powerful type theories (*e.g.* cubical type theory) would eliminate the necessity to use axioms, but it would not solve the transport hell problem. The *pair* example works so nicely, because we essentially turned the propositional equality into the definitional one.

3.1 Infinite Sequences

In the type theory proposed by Hickey, the only extension to the standard type theory is addition of very dependent *functions*. It is observed that (very) dependent records can be always presented as very dependent functions by choosing a domain type that enumerates the fields. This is essentially what the example with dependent pairs does – Σ type has two fields that are enumerated by booleans.

However, dependent functions can do more than that, as their domain does not have to be finite. Let us now consider such an infinite case by defining non-increasing infinite sequences. With a little abuse of notation, we can present those as the following very dependent type.

```
– ↓-seq : (n : Nat) → if n == 0 then ℕ else Fin (1 + ↓-seq (n - 1))
```

The same Münchhausen technique with forward declarations can be used to define such a function. We start by forward declaring Ty (expression on the right hand side of the arrow in the type above) and its interpretation I into natural numbers.

```
Ty : ℕ → Set
I : ∀ n → Ty n → ℕ
```

At the same time we forward declare the actual sequence that we want to define:

```
↓-seq : (n : ℕ) → Ty n
```

The type of the elements in the sequence is defined inductively as follows: for zero we have ℕ, the successor case gives us Fin of whatever the interpretation of the sequence that we are defining at predecessor is going to return us.

```
Ty 0       = ℕ
Ty (suc n) = Fin $ suc $ I n (↓-seq n)
```

The interpretation of the elements at the given sequence is straight-forward: zero case has a natural number that we return; elements of Fin types are casted into natural numbers.

```
I 0 n = n
I (suc n) i = toℕ i
```

Finally, we define the actual data of our non-increasing infinite sequence.

```
↓-seq 0 = 5
↓-seq 1 = # 3
↓-seq 2 = # 2
↓-seq (suc (suc (suc n))) = # 0
```

Notice that in this particular case, the types of the elements in sequence only depend on the previous element. We can imagine full induction, where the element can depend on all the previously defined elements. In this case induction-recursion becomes crucially important to generate an n-fold dependent type.

4 Multi-dimensional Arrays

The next example we consider is a type for multi-dimensional arrays that are commonly found in array languages such as APL [13]. Arrays can be thought of as n-dimensional rectangles, where the size of the rectangle is given by the *shape*, which is a vector of natural numbers describing extents along each dimension. Array languages follow the slogan "everything is an array", treating natural numbers and shape vectors as arrays. Natural numbers are 0-dimensional arrays, *e.g.* their shape is the empty vector. Shape vectors are 1-dimensional arrays, *e.g.* their shapes are 1-element vectors.

The problem with capturing this construction with inductive types is the following circularity. Array types depend on shapes, but the shapes are arrays. That is, the index of the type is the very type that we are defining.

4.1 Unshaped arrays

One way to define the array type inductively is to avoid the shape argument entirely. This construction is proposed by Jenkins [9]:

```
module Unshaped where
  data Ar : Set where
    z       : Ar            - Natural numbers with zero (z)
    s       : Ar → Ar       -   and successor (s)
    []      : Ar            - Cons lists with empty list ([])
    _::_    : Ar → Ar → Ar -   and cons operation (_::_).
    reshape : Ar → Ar → Ar - Multi-dimensional array constructor.
```

With these definitions we get a closed universe of arrays of natural numbers. On the positive side, we obtained the uniformity of arrays as in APL – if a function expects an array, it is possible to pass a number or a vector without any casting.

```
0_a 1_a 2_a 3_a : Ar       - Natural numbres
0_a = z; 1_a = s 0_a; 2_a = s 1_a; 3_a = s 2_a

v_2 v_4 mat_22 : Ar
v_2 = 2_a :: 2_a :: []     - Vector [2,2]
v_4 = 1_a :: 0_a           - Flattened identity matrix [[1,0],[0,1]]
  :: 0_a :: 1_a :: []
mat_22 = reshape v_2 v_4 - Identity matrix of shape [2,2]
```

On the negative side, our array type does not enforce any shape invariants. That is, we can produce non-rectangular arrays such as:

```
weird_1 = reshape (2_a :: 2_a :: []) (1_a :: 2_a :: 3_a :: [])
weird_2 = (3_a :: []) :: weird_1
```

While it might be possible to define the meaning for such cases, normally they are considered type errors. We could also try restricting these constructions with refinement types, but we are interested in intrinsically-typed solution instead.

4.2 Inductive-inductive

Intrinsically-typed array universe can be defined using inductive-inductive types, following the ideas from [18]. We define arrays and shapes mutually.

```
module Univ where
  data Sh : Set
  data Ar : Sh → Set
  data Sh where
    scal  : Sh
    vec   : Ar scal → Sh
    mda : ∀ {s} → Ar (vec s) → Sh
```

The shapes form the following hierarchy: scalars (*e.g.* natural numbers) have a unit shape; vector shapes are parametrised by scalars; multi-dimensional shapes are parametrised by vectors.

```
  Nat : Set
  Nat = Ar scal

  Vec : Nat → Set
  Vec n = Ar (vec n)

  prod : ∀ {n} → Vec n → Nat
```

We define names Nat and Vec which are synonyms for arrays of the corresponding shape. We also make a forward declaration of the *prod* function that computes the product of the given vector. Now we are ready to define the array universe as follows:

```
  data Ar where
    z        : Nat
    s        : Nat → Nat
```

```
[]      : Vec z
_::_    : ∀ {n} → Nat → Vec n → Vec (s n)
reshape : ∀ {n} → (s : Vec n) → Vec (prod s) → Ar (mda s)
```

We use exactly the same constructors as before, except vectors are indexed by their length, and multi-dimensional arrays are indexed by shape vectors. Also, the reshape constructor has a coherence condition saying that the number of elements (*prod s*) in the vector we are reshaping matches the new shape *s*.

We complete the definition of *prod*, expressing it as a fold with multiplication $\_*_n\_$ (defined as usual, not shown here).

```
0ₐ 1ₐ 2ₐ 3ₐ : Nat
0ₐ = z; 1ₐ = s z; 2ₐ = s 1ₐ; 3ₐ = s 2ₐ; 4ₐ = s 3ₐ

prod [] = 1ₐ
prod (x :: xs) = x *ₙ prod xs
```

Vector and matrix examples can be expressed as follows.

```
v₂ : Vec 2ₐ
v₂ = 2ₐ :: 2ₐ :: []

v₄ : Vec 4ₐ
v₄ = 1ₐ :: 0ₐ
   :: 0ₐ :: 1ₐ :: []

mat₂₂ : Ar (mda v₂)
mat₂₂ = reshape v₂ v₄
```

While the numbers, vectors and arrays are elements of the same universe, we did not achieve the desired array uniformity. The problem is that we maintain the distinction between arrays and shapes, even though morally they are the same thing. For example, the type of mat_{22} is Ar (mda v_2), not Ar v_2. Also, the expression reshape [] (1_a :: [])) cannot be typed as Nat, even though it is an array of the empty shape.

4.3 Münchhausen universe

In order to resolve the lack of uniformity, we use the Münchhausen method (the variant with forward declarations). Our goal is to equate Ar and Sh. Therefore, we forward-declare Sh as a placeholder to bootstrap the array type. After that we close the cycle by defining Sh to be Ar with a certain index.

We start with forward-declaring types N (natural numbers) and shapes Sh that are indexed by natural numbers. Both of these types are placeholders that we will eliminate later.

```
module UniformUniv where
  N : Set
  Sh : N → Set
```

The array type is a concrete definition, whereas its parameter is a placeholder type.

```
  data Ar : ∀ {n} → Sh n → Set
```

We make forward declarations of N and Sh constructors that are needed to fill-in the indices of the array type. Note that N constructors are used to fill-in the indices of Sh constructors.

```
z' : N
s' : N → N
[]' : Sh z'
_::'_ : ∀ {n} → N → Sh n → Sh (s' n)
```

We define names Nat and Vec for 0-dimensional and 1-dimensional arrays correspondingly. We also make a forward declaration of *prod* as before, except we use placeholder types.

```
Nat : Set
Nat = Ar []'

Vec : N → Set
Vec n = Ar (n ::' []')

prod : ∀ {n} → Sh n → N
```

Now we can define an array universe, exactly as before.

```
data Ar where
  z       : Nat
  s       : Nat → Nat
  []      : Vec z'
  _::_    : ∀ {n} → Nat → Vec n → Vec (s' n)
  reshape : ∀ {n} → (s : Sh n) → Vec (prod s) → Ar s
```

Finally, we eliminate the placeholder types by equating N and Sh with 0-dimensional and 1-dimensional arrays correspondingly. After we do this, we define the placeholder constructors to be those defined in Ar.

```
N    = Ar []'
Sh n = Ar (n ::' []')
z' = z; s' = s; []' = []; _::'_ = _::_
```

This closes the cycle and turns Ar into a very dependent type that is witnessed by the following Agda expression:

```
_ : ∀ {n : Ar []} → (s : Ar (n :: [])) → Set
_ = Ar
```

As expected, our examples are definable, and 1-dimensional arrays can be immediately used as array shapes.

```
0ₐ 1ₐ 2ₐ 3ₐ 4ₐ : Nat
0ₐ = z; 1ₐ = s 0ₐ; 2ₐ = s 1ₐ; 3ₐ = s 2ₐ; 4ₐ = s 3ₐ

v₂ : Ar (2ₐ :: [])
v₂ = 2ₐ :: 2ₐ :: []

v₄ : Ar (4ₐ :: [])
v₄ = 1ₐ :: 0ₐ :: 0ₐ :: 1ₐ :: []
```

One technical drawback that we ran into is that with such a cyclic type, Agda loops when attempting to define pattern-matching functions. The loop happens when solving the unification problem – it has to check that two arrays type match. As array types are indexed, Agda has to unify the indices, which triggers unifying the type of the indices, and so on. As a workaround, we can define eliminators via rewrite rules, which makes it possible to define *prod*. The rest works as expected.

```
prod {n} xs = sh-elim _ 1_a (λ n x xs r → x *_n r) n xs

mat_22 : Ar v_2
mat_22 = reshape v_2 v_4

scal-test : Nat
scal-test = reshape [] (1_a :: [])
```

5 Russell Universes

In pure type systems [5] there is no separate sort for terms and types, there are only terms and those terms which appear on the right hand side of the colon in the typing relation are called types. Using well-typed terms, this would lead to the following very dependent type for the sort of terms: Tm : (Γ : Con) → Tm Γ U → Set. That is, terms depend on a context and a term of type U. Using the Münchhausen method (its variant with forward declarations), we can make sense of this. We temporarily introduce types and the type U' for the universe, then after declaring the sort of terms we can say that actually types are just terms of type U', then we can add the actual U operator for terms and close the loop by saying that U' is the same as U. Using forward declarations, part of the syntax of type theory is given as follows.

```
data Con : Set
Ty : Con → Set    - forward declaration
data Con where
  •     : Con
  _▷_ : (Γ : Con) → Ty Γ → Con
U' : ∀ {Γ} → Ty Γ - forward declaration
data Tm : (Γ : Con) → Ty Γ → Set
Ty Γ = Tm Γ U'
data Tm where
  U    : ∀ {Γ} → Ty Γ
  Π    : ∀ {Γ} → (A : Ty Γ) → Ty (Γ ▷ A) → Ty Γ
  lam  : ∀ {Γ A B} → Tm (Γ ▷ A) B → Tm Γ (Π A B)
U' = U
```

Note that such a theory is inconsistent through Russell's paradox, but it is easy to fix this by stratification (adding natural number indices to Ty and U, see e.g. [15]). More precisely, we say that a stratified category with families (CwF [6]) with a type former U : (i : ℕ) → Ty Γ (i+1) satisfying U i [σ]T = U i is Russell if the equations Ty Γ i = Tm Γ (U i) and A [σ]T = A [σ]t hold (where _[_]T and _[_]t are the substitution operations for types and terms, respectively).

Any CwF with a hierarchy of Tarski universes can be equipped with a Russell family structure supporting the same type formers as the Tarski universe. A hierarchy of Tarski universes is given by the universe types $\mathsf{U}\ \mathsf{i} : \mathsf{Ty}\ \Gamma\ (\mathsf{i}{+}1)$, their decoding $\mathsf{El} : \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i}) \to \mathsf{Ty}\ \Gamma\ \mathsf{i}$, a code for each universe $\mathsf{u}\ \mathsf{i} : \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ (\mathsf{i}{+}1))$ such that their decoding is the actual universe $\mathsf{El}\ (\mathsf{u}\ \mathsf{i}) = \mathsf{U}\ \mathsf{i}$. We further have the evident substitution rules and additional operations expressing that U is closed under certain type formers. The Russell family structure then is defined by $\mathsf{Ty}^{\mathsf{R}}\ \Gamma\ \mathsf{i} := \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i})$ and $\mathsf{Tm}^{\mathsf{R}}\ \Gamma\ \mathsf{A} := \mathsf{Tm}\ \Gamma\ (\mathsf{El}\ \mathsf{A})$, both substitution operations are $\_[\_]\mathsf{t}$, context extension is $\Gamma \triangleright^{\mathsf{R}} \mathsf{A} := \Gamma \triangleright \mathsf{El}\ \mathsf{A}$. The Russell universe is defined as $\mathsf{U}^{\mathsf{R}}\ \mathsf{i} := \mathsf{u}\ \mathsf{i}$, thus we obtain the Russell sort equation by $\mathsf{Ty}^{\mathsf{R}}\ \Gamma\ \mathsf{i} = \mathsf{Tm}\ \Gamma\ (\mathsf{U}\ \mathsf{i}) = \mathsf{Tm}\ \Gamma\ (\mathsf{El}\ (\mathsf{u}\ \mathsf{i})) = \mathsf{Tm}^{\mathsf{R}}\ \Gamma\ (\mathsf{U}^{\mathsf{R}}\ \mathsf{i})$. We formalised this model construction using the shallow embedding trick of [14], the formalisation is part of the source code of the current paper[8].

6 Type Theory without Contexts

Having Russell universes could be called "type theory without types" as types are just special terms. Type theory without contexts is when contexts are just types without free variables.

When defining type theory as an algebraic theory, the final goal is to describe the rules for types and terms. Contexts and substitutions (the category structure) are only there as supporting infrastructure. However, when enough structure is added to types and terms, we don't need this supporting infrastructure anymore and we can get rid of it using the Münchhausen technique. We will still have explicit substitutions, but we use terms instead of context morphisms. The resulting theory with very dependent types includes the following sorts and operations. Note that some of these operations are not only very dependently typed, but the typing is very mutual: for example, the type of Ty includes $\top$ which is only listed later.

$$
\begin{array}{ll}
\mathsf{Ty} & : \mathsf{Ty}\ \top \to \mathsf{Set} \\
\mathsf{Tm} & : (\Gamma : \mathsf{Ty}\ \top) \to \mathsf{Ty}\ \Gamma \to \mathsf{Set} \\
\_[\_]\mathsf{T} & : \mathsf{Ty}\ \Gamma \to \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T}) \to \mathsf{Ty}\ \Delta \\
\_[\_]\mathsf{t} & : \mathsf{Tm}\ \Gamma\ \mathsf{A} \to (\sigma : \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T})) \to \mathsf{Tm}\ \Delta\ (\mathsf{A}\ [\ \sigma\ ]\mathsf{T}) \\
\mathsf{id} & : \mathsf{Tm}\ \Gamma\ (\Gamma\ [\ \mathsf{tt}\ ]\mathsf{T}) \\
\top & : \mathsf{Ty}\ \Gamma \\
\mathsf{tt} & : \mathsf{Tm}\ \Gamma\ \top \\
\Sigma & : (\mathsf{A} : \mathsf{Ty}\ \Gamma) \to \mathsf{Ty}\ (\Sigma\ \Gamma\ (\mathsf{A}\ [\ \mathsf{snd}\ \mathsf{id}\ ]\mathsf{T})) \to \mathsf{Ty}\ \Gamma \\
\_,\_ & : (\mathsf{a} : \mathsf{Tm}\ \Gamma\ \mathsf{A}) \to \mathsf{Tm}\ \Gamma\ (\mathsf{B}\ [\ \mathsf{id}\ ,\ \mathsf{a}\ ]\mathsf{T}) \to \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B}) \\
\mathsf{fst} & : \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B}) \to \mathsf{Tm}\ \Gamma\ \mathsf{A} \\
\mathsf{snd} & : (\mathsf{w} : \mathsf{Tm}\ \Gamma\ (\Sigma\ \mathsf{A}\ \mathsf{B})) \to \mathsf{Tm}\ \Gamma\ (\mathsf{B}\ [\ \mathsf{id}\ ,\ \mathsf{fst}\ \mathsf{w}\ ]\mathsf{T})
\end{array}
$$

It is difficult to derive the above in Agda using forward declarations or rewrite rules, but working on paper (in extensional type theory) this is possible. A *model of type theory without contexts* is given by a CwF with $\top$ and Σ types[9] where the following equations hold.

[8] See `https://bitbucket.org/akaposi/combinator/src/master/post-types2022/russel.lagda`

[9] List of notations: the category is denoted Con, Sub, $\_\circ\_$, id, the empty context (terminal object) $\diamond$, the empty substitution ε, types are $\mathsf{Ty}\ \Gamma$ with the substitution operation $\_[\_]\mathsf{T}$, terms $\mathsf{Tm}\ \Gamma\ \mathsf{A}$ with $\_[\_]\mathsf{t}$, context extension $\_\triangleright\_$, substitution extension $\_,\_$ and projections $\mathsf{p} : \mathsf{Sub}\ (\Gamma \triangleright \mathsf{A})\ \Gamma$, $\mathsf{q} : \mathsf{Tm}\ (\Gamma \triangleright \mathsf{A})\ (\mathsf{A}\ [\ \mathsf{p}\ ]\mathsf{T})$. The type former $\top : \mathsf{Ty}\ \Gamma$ comes with constructor tt and η law. Σ's constructor is denoted $\_,\_$, the destructors are fst and snd and we have both β laws and an η law.

$$
\begin{aligned}
&\mathsf{Con} && = \mathsf{Ty}\ \diamond \\
&\mathsf{Sub}\ \Delta\ \Gamma && = \mathsf{Tm}\ \Delta\ (\Gamma\ [\ \varepsilon\ ]\mathsf{T}) \\
&\sigma \circ \nu && = \sigma\ [\ \nu\ ]\mathsf{t} \\
&\diamond && = \top \\
&\varepsilon && = \mathsf{tt} \\
&\Gamma \triangleright \mathsf{A} && = \Sigma\ \Gamma\ (\mathsf{A}\ [\ \mathsf{q}\ ]\mathsf{T}) \\
&\sigma\ ,\ \mathsf{t} && = \sigma\ ,\ \mathsf{t} \\
&\mathsf{p} && = \mathsf{fst}\ \mathsf{id} \\
&\mathsf{q} && = \mathsf{snd}\ \mathsf{id}
\end{aligned}
$$

Note that the well-typedness of the second equation depends on the first equation, as $\Gamma : \mathsf{Con}$ has to be viewed as $\Gamma : \mathsf{Ty}\ \diamond$ and then we can substitute it with the empty substitution to obtain $\Gamma\ [\ \varepsilon\ ]\mathsf{T} : \mathsf{Ty}\ \Delta$. Just as substitutions are special terms, composition of substitutions is a special case of substitution of terms, the empty context is $\top : \mathsf{Ty}\ \diamond$, context extension is a Σ type where $\mathsf{A} : \mathsf{Ty}\ \Gamma$, but Σ requires a $\mathsf{Ty}\ (\diamond \triangleright \Gamma)$, so we need a $\mathsf{Sub}\ (\diamond \triangleright \Gamma)\ \Gamma = \mathsf{Tm}\ (\diamond \triangleright \Gamma)\ (\Gamma\ [\ \varepsilon\ ]\mathsf{T}) = \mathsf{Tm}\ (\diamond \triangleright \Gamma)\ (\Gamma\ [\ \mathsf{p}\ ]\mathsf{T})$, which is given by $\mathsf{q}\ \{\diamond\}\{\Gamma\}$.

We can check that in a model of type theory without contexts, the very dependent types listed above are all valid.

As for Russell models, we have a model construction which replaces any CwF with $\top$ and Σ with a model without contexts. We cannot directly use the equations of model without contexts above for the model construction. *E.g.* if we said that $\mathsf{Con}' := \mathsf{Ty}\ \diamond$ and $\diamond' := \top$ and $\mathsf{Ty}'\ \Gamma := \mathsf{Ty}\ (\diamond \triangleright \Gamma)$ then we would have $\mathsf{Con}' = \mathsf{Ty}\ \diamond \neq \mathsf{Ty}\ (\diamond \triangleright \top) = \mathsf{Ty}'\ \diamond'$. Instead we define $\mathsf{Con}' := \mathsf{Ty}\ (\diamond \triangleright \top)$, $\diamond' := \top$ and $\mathsf{Ty}'\ \Gamma := \mathsf{Ty}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})$. Now we have $\mathsf{Con}' = \mathsf{Ty}\ (\diamond \triangleright \top) = \mathsf{Ty}\ (\diamond \triangleright \top\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T}) = \mathsf{Ty}\ (\diamond \triangleright \diamond'\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T}) = \mathsf{Ty}'\ \diamond'$. We refer to Appendix A for the definition of the rest of the components of the output model and also for a proof that if the input model has Π types then so does the output model. We formalised this model construction using shallow embedding [14], the formalisation is part of the source code of the current paper[10].

7 Combinatory Type Theory

In our final example, we also present a (dependnet) type theory without contexts. Instead of eliminating contexts with equations as we did in the previous section, we avoid introducing them in the encoding. This raises the question: if there are no contexts, how do we talk about well-scoped variables? As a matter of fact, we do not talk about variables at all.

It is well known that for simply-typed systems, combinator calculus [11] gives a contextless presentation of the type system. There are no variables, function space is built-in, and the combinators S and K are used to define functions.

Combinator calculus for dependently-typed systems is a much more challenging [17] task, and it was never defined. Unsurprisingly, contextless dependently-typed theory is an example of a very dependent type, and we use Münchhausen method to define it. Specifically, we use postulates and rewrite rules to encode very dependent types. While there might be a solution with forward declarations, we chose rewrite rules for the sake of simplicity of the presentation.

[10] See `https://bitbucket.org/akaposi/combinator/src/master/post-types2022/uncat.lagda`

First Attempt

In our first attempt we started defining a non-dependent function type, hoping to internalise it in the universe and define Π types afterwards. Concretely, we define types Ty, terms Tm indexed by types, the universe U, and eliminate the Ty type. After that, we define the arrow type _⇒_, applications, the arrow type within the universe |⇒| and the equation that turns the arrow into the internal arrow.

```
module FirstAttempt where
  postulate
    Ty  : Set                              - Types
    Tm  : Ty → Set                         - Terms
    U   : Ty                               - Universe
    TmU : Ty ≡ Tm U                        - Russell-ification
    {-# REWRITE TmU #-}
    _⇒_ : Ty → Ty → Ty                     - Non-dependent (external) arrow type
    _$_ : Tm (X ⇒ Y) → Tm X → Tm Y         - Applications
    |⇒| : Tm (U ⇒ U ⇒ U)                   - Internal arrow type
    ∅⇒  : X ⇒ Y ≡ |⇒| $ X $ Y              - Internalising arrow
```

While this looks promising, after rewriting ∅⇒ we run into the following problem. Consider the sequence of rewrites that is happening for the type Tm $(X \Rightarrow Y)$ which is the type of the first argument of the application _$_.

```
_1 : Tm (X ⇒ Y)                          - Expands to
_1 : Tm (|⇒| $ X $ Y)                    - Show hidden arguments
_2 : Tm ((_$_ {U}{U ⇒ U} |⇒| X) $ Y)     - Arrow in (U ⇒ U) again!
```

As Agda applies all the rewrite rules before type checking, we end-up in the infinite rewrite loop. There does not seem to be an easy fix.

Second Attempt

Now we start with Π types straight away and use them to define dependent combinators. The notion of types, terms and the universe is the same as before.

```
module SecondAttempt where
  postulate
    Ty : Set
    Tm : Ty → Set
    U : Ty
    Tm-U : Tm U ≡ Ty
    {-# REWRITE Tm-U #-}
```

We introduce the notion of a U-valued family and the application operation for it. Using family we can immediately define Π types and applications to the terms of Π types.

```
    Fam  : Ty → Ty                          - Fam X ≈ (X ⇒ U)
    _$f_ : Tm (Fam X) → Tm X → Ty           - Apply (x : X) to (t : Fam X)

    Pi   : (X : Ty) → Tm (Fam (Fam X))      - X → ((X ⇒ U) ⇒ U)
    _$_  : {X : Ty}{Y : Tm (Fam X)}
         → Tm (Pi X $f Y) → (a : Tm X) → Tm (Y $f a)
```

Consider defining a non-dependent function type for (X Y : Ty) using the Pi type. We can immediately apply X to Pi, but then we need to turn Y into a constant X-family in order to complete the definition (Pi X $f □). To achieve this we introduce the Kf combinator that turns a type into a constant family. Its beta rule is the same as of the standard K combinator. Using Kf we can complete the definition of non-dependent arrow.

```
  Kf    : (Y : Ty) → Tm (Fam X) - Kf Y ≈ λ a → Y
  Kf$   : ∀ {a : Tm X} → _$f_ {X} (Kf Y) a ≡ Y
  {-# REWRITE Kf$ #-}

_⇒_ : (X Y : Ty) → Ty
X ⇒ Y = Pi X $f (Kf Y)
```

Let us remind ourselves, the type of dependent K combinator:

```
[K] : (X : Set)(Y : X → Set) → (x : X) (y : Y x) → X
[K] X Y x y = x
```

Translation into our formalism requires expressing (x : X)(y : Y x) → X as a Pi type. More precisely, how do we express (Y x → X) as an X-family? We do this by introducing a helper combinator with the corresponding beta rule. After that, defining dependent K and its beta rule becomes straight-forward.

```
postulate - Dependent K
  Yx⇒Z : ∀ X (Y : Tm (Fam X)) → (Z : Ty) → Tm (Fam X)
  Yx⇒Z$ : ∀ X Y Z {x : Tm X} → Yx⇒Z X Y Z $f x ≡ Y $f x ⇒ Z
  {-# REWRITE Yx⇒Z$ #-}

  Kd : {Y : Tm (Fam X)} → Tm (Pi X $f Yx⇒Z X Y X)
  Kd$ : ∀ {Y : Tm (Fam X)}{x : Tm X}{y : Tm (Y $f x)}
      → Kd {X = X}{Y = Y} $ x $ y ≡ x
  {-# REWRITE Kd$ #-}
```

Similarly to dependent K, we start with reminding ourselves the type of the dependent S combinator. We will use the same strategy of defining extra combinators to construct parts of the type signature.

```
[S] : (X : Set)(Y : X → Set)
        (Z : (x : X) → Y x → Set)           - λ (x : X) → Yx⇒U x
   → (f : (x : X) → (y : Y x) → Z x y) - λ (x : X) → Π[Yx][Zx] x
   → (g : (x : X) → Y x)
   → ((x : X) → Z x (g x))             - λ (g : ΠXY) → ΠX[Zx[gx]] g
[S] X Y Z f g x = f x (g x)
```

We annotate the combinators we introduced at the corresponding positions of the [S] type. With these definitions, we can define dependent S and its beta rule as follows.

```
postulate - Dependent S
  Yx⇒U : ∀ X (Y : Tm (Fam X)) → Tm (Fam X)
  Yx⇒U$ : ∀ X Y {x : Tm X} → Yx⇒U X Y $f x ≡ Fam (Y $f x)
  {-# REWRITE Yx⇒U$ #-}

  Π[Yx][Zx]  : ∀ X (Y : Tm (Fam X)) → (Z : Tm (Pi X $f Yx⇒U X Y)) → Tm (Fam X)
  Π[Yx][Zx]$ : ∀ X Y Z {x : Tm X} → Π[Yx][Zx] X Y Z $f x ≡ Pi (Y $f x) $f (Z $ x)
  {-# REWRITE Π[Yx][Zx]$ #-}

  Zx[gx] : ∀ X (Y : Tm (Fam X)) (Z : Tm (Pi X $f Yx⇒U X Y))
          → Tm (Pi X $f Y) → Tm (Fam X)
  Zx[gx]$ : ∀ X Y Z g {x} → Zx[gx] X Y Z g $f x ≡ Z $ x $f (g $ x)
  {-# REWRITE Zx[gx]$ #-}

  ΠX[Zx[gx]] : ∀ X (Y : Tm (Fam X)) (Z : Tm (Pi X $f Yx⇒U X Y))
               → Tm (Fam (Pi X $f Y))
  ΠX[Zx[gx]]$ : ∀ X Y Z g → ΠX[Zx[gx]] X Y Z $f g ≡ Pi X $f (Zx[gx] X Y Z g)
  {-# REWRITE ΠX[Zx[gx]]$ #-}

  Sd : {Y : Tm (Fam X)}{Z : Tm (Pi X $f Yx⇒U X Y)}
     → Tm (Pi X $f Π[Yx][Zx] X Y Z
               ⇒ (Pi (Pi X $f Y) $f (ΠX[Zx[gx]] X Y Z)))
  Sd$ : {Y : Tm (Fam X)}{Z : Tm (Pi X $f Yx⇒U X Y)}
      → {f : Tm (Pi X $f Π[Yx][Zx] X Y Z) }
      → {g : Tm (Pi X $f Y)}
      → {x : Tm X}
      → Sd $ f $ g $ x ≡ f $ x $ (g $ x)
  {-# REWRITE Sd$ #-}
```

Finally, with a few more rewrite rules, we can define non-dependent S and K combinators as special cases of their dependent versions.

```
K : Tm (X ⇒ Y ⇒ X)
K {X}{Y} = Kd {X}{Kf Y}

S : Tm ((X ⇒ Y ⇒ Z) ⇒ (X ⇒ Y) ⇒ X ⇒ Z)
S {X}{Y}{Z} = Sd {X}{Kf Y}{K $ (Kf Z)}
```

We made a good progress with defining combinatory type theory. However, current combinators are not yet powerful enough to internalise Pi and Fam. The problem is that in Pi, Kd and Sd type parameters X, Y and Z are quantified externally. We need to define the version of these combinators that internalises this quantification within U. There is no conceptual problem in doing so, but the resulting terms become incredibly large and inconvenient to work with. Specifically, the one for the dependent S combinator. It is not clear whether there is a more elegant way of doing this.

8 Conclusions

This paper demonstrates a technique to justify and make practical use of very dependent types. Our method is based on the observation that the "cycle" of a very dependent type can be "cut" by introducing placeholder types, defining the data and then eliminating placeholders by means of equations.

When we try to apply the proposed technique within the actual theorem provers such as Agda, we have a few choices on how to implement this. First, we can pack together placeholders, data and explicit equalities, *e.g.* as we do in DPair type in Section 3. This is a straight-forward implementation of the Münchhausen technique. However, dealing with explicit propositional equalities as parts of data often brings us to the situation called "transport hell". For example, the isomorphism proof about DPairs is an instance of that. Alternatively, for the objects of very dependent types, we can turn propositional equalities into definitional ones. On paper, extensional type theory achieves this, and in special cases we can use shallow embedding (as in the formalisation of Sections 5 and 6). In Agda, there are two ways to do this: forward declarations and rewrite rules. Forward declarations are demonstrated when declaring pair in Section 3, Ar universe in Section 4 and Tm in Section 5. While this is a very convenient feature of Agda, it is considered[11] not very well understood by many Agda developers. Also, as we have seen with Ar example, currently it leads to loops in the typechecker, which is clearly a bug.

Rewrite rules [7] make it possible to turn arbitrary propositional equalities into definitional ones, but this feature of Agda is considered unsafe. However, it is clearly a localized implementation of extensional type theory which is conservative over intensional type theory with extensionality principles (as available in Cubical Agda). We expect that the conservativity result [12] extends to our setting and hence the use of rewriting rules is only a cosmetic and labour saving tool to avoid *transport hell*. We use rewrite rules in Section 7. Currently, the interplay between the rewrite rules and the typechecker is not always satisfying. For example, our first attempt in Section 7 ends up in an infinite rewrite, as all the rules have to fire before the typechecker. We believe that more interleaved approach to rewriting could make our example to typecheck.

The examples show that very dependent types can be used in a fully algebraic setting, *i.e.* without referring to untyped preterms as in [10]. The essential ingredient are forward declarations, *i.e.* we introduce the type of an object but only define it later while already using it in the types of other objects – see [2] for a formal definition of this concept. This is also the idea in inductive-inductive definitions, where constructors may depend on previous constructors [4, 8].

Clearly, Agda provides us with a mechanism to play around with these concepts but it is not yet clear what exactly the theory behind these constructions is. In this sense, our paper raises questions instead of answering them. We believe that this is a valuable contribution to the subject.

References

1 Agda Development Team. *Agda 2.6.3 documentation*, 2023. Accessed [2023/05/01]. URL: `https://agda.readthedocs.io/en/v2.6.3/`.

2 Thorsten Altenkirch, Nils Anders Danielsson, Andres Löh, and Nicolas Oury. ΠΣ: Dependent types without the sugar. *Functional and Logic Programming*, pages 40–55, 2010.

[11] See the following Agda issue `https://github.com/agda/agda/issues/1556` that discusses forward declarations and very dependent types.

3 Thorsten Altenkirch, Ambrus Kaposi, Artjoms Šinkarovs, and Tamás Végh. The Münchhausen method and combinatory type theory. In Delia Kesner and Pierre-Marie Pédrot, editors, *28th International Conference on Types for Proofs and Programs (TYPES 2022)*. University of Nantes, 2022. URL: https://types22.inria.fr/files/2022/06/TYPES_2022_paper_8.pdf.

4 Thorsten Altenkirch, Peter Morris, Fredrik Nordvall Forsberg, and Anton Setzer. A categorical semantics for inductive-inductive definitions. In *CALCO*, pages 70–84, 2011. doi:10.1007/978-3-642-22944-2_6.

5 Henk Barendregt. Introduction to generalized type systems. *J. Funct. Program.*, 1(2):125–154, 1991. doi:10.1017/s0956796800020025.

6 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019. arXiv:1904.00827.

7 Jesper Cockx. Type Theory Unchained: Extending Agda with User-Defined Rewrite Rules. In Marc Bezem and Assia Mahboubi, editors, *25th International Conference on Types for Proofs and Programs (TYPES 2019)*, volume 175 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 2:1–2:27, Dagstuhl, Germany, 2020. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. doi:10.4230/LIPIcs.TYPES.2019.2.

8 Fredrik Nordvall Forsberg. *Inductive-inductive definitions.* PhD thesis, Swansea University (United Kingdom), 2013.

9 J. I. Glasgow and M. A. Jenkins. Array theory, logic and the nial language. In *Proceedings. 1988 International Conference on Computer Languages*, pages 296–303, October 1988. doi:10.1109/ICCL.1988.13077.

10 Jason J. Hickey. Formal objects in type theory using very dependent types. In *In Foundations of Object Oriented Languages 3*, 1996.

11 J. Roger Hindley and Jonathan P. Seldin. *Introduction to Combinators and Lambda-Calculus.* Cambridge University Press, 1986.

12 Martin Hofmann. Conservativity of equality reflection over intensional type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs*, pages 153–164, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

13 Kenneth E. Iverson. *A Programming Language.* John Wiley & Sons, Inc., New York, NY, USA, 1962.

14 Ambrus Kaposi, András Kovács, and Nicolai Kraus. Shallow embedding of type theory is morally correct. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 329–365, Cham, 2019. Springer International Publishing.

15 András Kovács. Generalized universe hierarchies and first-class universe levels. In Florin Manea and Alex Simpson, editors, *30th EACSL Annual Conference on Computer Science Logic, CSL 2022, February 14-19, 2022, Göttingen, Germany (Virtual Conference)*, volume 216 of *LIPIcs*, pages 28:1–28:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022. doi:10.4230/LIPIcs.CSL.2022.28.

16 Per Martin-Löf. An intuitionistic theory of types. In Giovanni Sambin and Jan M. Smith, editors, *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 127–172. Oxford University Press, 1998.

17 Conor McBride. What is the combinatory logic equivalent of intuitionistic type theory? Answer to question on StackOverflow, 2012. URL: https://stackoverflow.com/questions/11406786/what-is-the-combinatory-logic-equivalent-of-intuitionistic-type-theory.

18 Artjoms Šinkarovs. Multi-dimensional arrays with levels. In Max S. New and Sam Lindley, editors, *Proceedings Eighth Workshop on Mathematically Structured Functional Programming, MSFP@ETAPS 2020, Dublin, Ireland, 25th April 2020*, volume 317 of *EPTCS*, pages 57–71, 2020. doi:10.4204/EPTCS.317.4.

A The Type Theory without Contexts model construction

The input is a CwF with $\top$ and Σ types both having η rules. We use the same notation as in Section 6, the components of the model are Con, Ty, and so on. The components of the output model without contexts are denoted the same. We list all of them here in the following order: category, terminal object, types, terms, context extension, unit, Σ. This model construction was fully formalised in Agda.

$$\begin{aligned}
&\mathsf{Con} &&:= \mathsf{Ty}\ (\diamond \triangleright \top)\\
&\mathsf{Sub}\ \Delta\ \Gamma &&:= \mathsf{Tm}\ (\diamond \triangleright \Delta\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\ (\Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\\
&\sigma \circ \nu &&:= \sigma\ [\ \varepsilon\ ,\ \nu\ ]\mathsf{t}\\
&\mathsf{id} &&:= \mathsf{q}\\
&\diamond &&:= \top\\
&\varepsilon &&:= \mathsf{tt}\\
&\mathsf{Ty}\ \Gamma &&:= \mathsf{Ty}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\\
&\mathsf{A}\ [\ \sigma\ ]\mathsf{T} &&:= \mathsf{A}\ [\ \varepsilon\ ,\ \sigma\ ]\mathsf{T}\\
&\mathsf{Tm}\ \Gamma\ \mathsf{A} &&:= \mathsf{Tm}\ (\diamond \triangleright \Gamma\ [\ \varepsilon\ ,\ \mathsf{tt}\ ]\mathsf{T})\ \mathsf{A}\\
&\mathsf{t}\ [\ \sigma\ ]\mathsf{t} &&:= \mathsf{t}\ [\ \varepsilon\ ,\ \sigma\ ]\mathsf{t}\\
&\Gamma \triangleright \mathsf{A} &&:= \Sigma\ \Gamma\ (\mathsf{A}\ [\ \varepsilon\ ,\ \mathsf{q}\ ]\mathsf{T})\\
&\sigma\ ,\ \mathsf{t} &&:= \sigma\ ,\ \mathsf{t}\\
&\mathsf{p} &&:= \mathsf{fst}\ \mathsf{q}\\
&\mathsf{q} &&:= \mathsf{snd}\ \mathsf{q}\\
&\top &&:= \top\\
&\mathsf{tt} &&:= \mathsf{tt}\\
&\Sigma\ \mathsf{A}\ \mathsf{B} &&:= \Sigma\ \mathsf{A}\ (\mathsf{B}[\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ])\mathsf{T}\\
&\mathsf{u}\ ,\ \mathsf{v} &&:= \mathsf{u}\ ,\ \mathsf{v}\\
&\mathsf{fst}\ \mathsf{t} &&:= \mathsf{fst}\ \mathsf{t}\\
&\mathsf{snd}\ \mathsf{t} &&:= \mathsf{snd}\ \mathsf{t}
\end{aligned}$$

All the equations hold. If the input model has Π types, so does the output model. If the input model has a Coquand-universe, so does the output model. The operations are the following.

$$\begin{aligned}
&\Pi\ \mathsf{A}\ \mathsf{B} &&:= \Pi\ \mathsf{A}\ (\mathsf{B}[\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ])\mathsf{T}\\
&\mathsf{lam}\ \mathsf{t} &&:= \mathsf{lam}\ (\mathsf{t}\ [\ \varepsilon\ ,\ (\mathsf{q}\ [\ \mathsf{p}\ ]\mathsf{t}\ ,\ \mathsf{q})\ ]\mathsf{t})\\
&\mathsf{app}\ \mathsf{t} &&:= (\mathsf{app}\ \mathsf{t})\ [\ \varepsilon\ ,\ \mathsf{fst}\ \mathsf{q}\ ,\ \mathsf{snd}\ \mathsf{q}\ ]\mathsf{t}\\
&\mathsf{U} &&:= \mathsf{U}\\
&\mathsf{El}\ \mathsf{t} &&:= \mathsf{El}\ \mathsf{t}\\
&\mathsf{c}\ \mathsf{A} &&:= \mathsf{c}\ \mathsf{A}
\end{aligned}$$

All the equations hold.

Pragmatic Isomorphism Proofs Between Coq Representations: Application to Lambda-Term Families

Catherine Dubois ✉ 🏠
Samovar, ENSIIE, 1 square de la résistance, 91025 Évry-Courcouronnes, France

Nicolas Magaud ✉ 🏠
Lab. ICube UMR 7357 CNRS Université de Strasbourg, 67412 Illkirch, France

Alain Giorgetti ✉ 🏠
Université de Franche-Comté, CNRS, Institut FEMTO-ST, F-25030 Besançon, France

Abstract

There are several ways to formally represent families of data, such as lambda terms, in a type theory such as the dependent type theory of Coq. Mathematical representations are very compact ones and usually rely on the use of dependent types, but they tend to be difficult to handle in practice. On the contrary, implementations based on a larger (and simpler) data structure combined with a restriction property are much easier to deal with.

In this work, we study several families related to lambda terms, among which Motzkin trees, seen as lambda term skeletons, closable Motzkin trees, corresponding to closed lambda terms, and a parameterized family of open lambda terms. For each of these families, we define two different representations, show that they are isomorphic and provide tools to switch from one representation to another. All these datatypes and their associated transformations are implemented in the Coq proof assistant. Furthermore we implement random generators for each representation, using the QuickChick plugin.

2012 ACM Subject Classification Theory of computation → Lambda calculus; Theory of computation → Logic and verification; Theory of computation → Type theory

Keywords and phrases Data Representations, Isomorphisms, dependent Types, formal Proofs, random Generation, lambda Terms, Coq

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.11

Supplementary Material *Software (Source Code)*: `https://github.com/magaud/postTYPES2022`
archived at `swh:1:dir:f01d28048087438c173ad32a5536f79755285465`

Funding *Alain Giorgetti*: Supported by the EIPHI Graduate School (contract ANR-17-EURE-0002).

1 Introduction

Choosing the most appropriate implementation of mathematical objects to perform computations and proofs is challenging. Indeed, efficient (well-suited for computations) representations are often difficult to handle when it comes to proving properties of these objects. Conversely, well-suited representations for proofs often have fairly poor performances when it comes to computing. The simplest example is the implementation of natural numbers. Using a unary representation, proofs (especially inductive reasoning) are easy to carry out but computing is highly inefficient. Using a binary representation makes computations faster, however it is more difficult to use reasoning principles such as the induction principle on natural numbers.

In the field of λ-calculus, representations that are closest to mathematics are usually implemented using dependent types. This makes them easily readable and understandable by mathematicians. However it is rather challenging and requires a strong background in functional programming and theorem proving to handle them smoothly. Representations

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 11; pp. 11:1–11:19
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

based on a larger type (a non-dependent one) and a restriction property are easier to handle in practice, but less intuitive. In addition, one needs to take extra care to make sure that the combination of the larger type and the restriction property exactly represents the expected objects.

Overall there is no perfect representation for a given mathematical object. To overcome this challenge, we propose to deal with several different isomorphic representations of families of mathematical objects simultaneously. To do that, we present a rigorous methodology to partially automate the construction of the transformation functions between two isomorphic representations and prove these transformations correct. We apply these techniques to some families of objects related to λ-calculus, namely closable Motzkin trees, uniquely closable Motzkin trees and m-open λ-terms.

Our first examples revisit an article of Bodini and Tarau [3] in which they define Prolog generators for closed lambda terms and their skeletons seen as Motzkin trees, efficient generators for closable and uniquely closable skeletons and study their statistical properties. Our contributions are to formalize in Coq these different notions, prove the equivalence of several definitions that underlie the generators designed by Bodini and Tarau, and write random generators to be used with QuickChick [12]. We then extend the discourse to a parameterized family of open λ-terms, named m-open λ-terms. All the considered representations, transformations between isomorphic representations and isomorphism proofs are formalized[1] in the Coq proof assistant [2, 7]. We propose some generic tools to help setting up the correspondence between two isomorphic types more easily. We hope such a methodology could be reused to deal with other families of objects, having different and isomorphic representations.

Related Work. Dealing with various isomorphic representations of the same mathematical objects is a common issue in computer science. Research results span from theoretical high-level approaches such as homotopy type theory [19] or cubical type theory [5] to more pragmatic proposals such as ours. In the context of formal specifications and proofs about mathematical concepts, several frameworks have been proposed to deal with several types and their transformation functions. A seminal work on changing (isomorphic) data representation was implemented by Magaud [14] as a plugin for Coq in the early 2000s. In this approach, the transformation functions were provided by the user and only the proofs were ported. Here, we aim at helping the programmer to write the transformation functions as well as their proofs of correctness. In [6], Cohen et al. focus on refining from abstract representations, well-suited for reasoning, to computationally well-behaved representations. In our work, both representations are considered of equal importance, and none of them is preferred. Finally, our work is closely related to the concept of views, introduced by Wadler in [20] and heavily used in the dependently-typed programming language Epigram [16]. In this approach, operations are made independent of the actual implementation of the types they work on. Pattern-matching on an element of type A can be carried out following the structure of the type B provided A and B are isomorphic types. The correspondence functions we shall implement in this article provide an example of a concrete implementation of views.

Regarding random generators and enumerators, Paraskevopoulou et al. [17] recently proposed a new framework, on top of the QuickChick testing tool for Coq. It allows to automatically derive such generators by extracting the computational contents from inductive relations.

[1] The Coq code is available at `https://archive.softwareheritage.org/browse/origin/https://github.com/alaingiorgetti/postTYPES2022`.

Paper Outline. In Sect. 2, we present a general methodology and interfaces to capture all the features of two representations of a given family of objects, and to switch easily from one representation to the other. In Sect. 3, we show how our approach applies to representations of closable Motzkin trees – that are the skeletons of closed λ-terms – and to representations of uniquely closable Motzkin trees. In Sect. 4, we adapt our approach to the parameterized family of m-open λ-terms. In Sect. 5, several applications of the presented isomorphic types are exposed. In Sect. 6, we draw some conclusions and present some promising perspectives.

2 Specifying Families Using Two Different Representations

As we shall see with examples related to pure λ-terms, a family of mathematical objects can usually be defined formally in two different but equivalent ways: either using an inductive datatype, possibly dependent, or using a larger non dependent datatype, together with a restriction property. In this section, we summarize which elements are required to specify the two datatypes and their basic properties. We then show how to derive the isomorphism properties automatically. One of these isomorphism properties can always be derived automatically, using a generic approach based on a functor, whereas the other one, which relies on a proof by induction on the data, is carried out using Ltac. In the examples presented in this article, the types have at most one level of dependency. Even if the Ltac code is as generic as possible, it may not generalize well when the level and complexity of the dependencies increase.

2.1 Types

A *restricted type* (`T`,`is_P`) is a dependent pair defined by a type `T : Type`, called its *base type*, and a predicate `is_P : T → Prop`, called its *restriction* or *filter*. The restricted type (`T`,`is_P`) is intended to represent the inhabitants of `T` satisfying the restriction `is_P`. For practical reasons, these two objects are encapsulated together as a record type `rec_P` isomorphic to the Σ-type $\{x : \texttt{T} \mid \texttt{is\_P}\ x\}$.

```
Record rec_P := Build_rec_P {
  P_struct :> T;
  P_prop : is_P P_struct
}.
```

In addition to this practical type, we assume that we also have another possibly dependent type `P` for the same family of objects. This type is usually closer to the way mathematicians would define such objects. However, it may be less convenient to handle in practice (e.g. when proving in a proof assistant such as Coq) and thus we shall prefer using the larger type `T` and the restriction `is_P` rather than the type `P` when programming operations and proving lemmas on such a family.

2.2 Transformations and Their Properties

Once the datatypes `T` and `P` and the filter `is_P` are defined, we build the expected isomorphisms as two transformation functions `rec_P2P` (from `rec_P` $\equiv \{x : \texttt{T} \mid \texttt{is\_P}\ x\}$ to `P`) and `P2rec_P` (from `P` to `rec_P` $\equiv \{x : \texttt{T} \mid \texttt{is\_P}\ x\}$). The first function `rec_P2P` can be defined as follows, with an auxiliary function `T2P : ∀ (x:T), is_P x → P` transforming any element x : `T` that satisfies `is_P` into an element of `P`.

```
Definition rec_P2P m := T2P (P_struct m) (P_prop m).
```

To define the reverse transformation P2rec_P, we first implement a function P2T from P to T and then prove that the image of any x by P2T satisfies the predicate is_P, i.e. we prove the following lemma:

```
Lemma is_P_lemma: ∀ v, is_P (P2T v).
```

Then the transformation P2rec_P can be defined as follows:

```
Definition P2rec_P (x:P) : rec_P := Build_rec_P (P2T x) (is_P_lemma x).
```

2.3 Partial Automation of Specification and Proofs

In order to automate some parts of the process, we provide an abstract definition of the minimum requirements for the two involved types, as shown in the module type declaration (a.k.a. *interface* or *signature*) family reproduced in the following code snippet.

```
Module Type family.
  Parameter T : Set.
  Parameter is_P : T → Prop.
  Parameter P : Set.
  Parameter T2P : ∀ (x:T), is_P x → P.
  Parameter P2T : P → T.
  Parameter is_P_lemma : ∀ v, is_P (P2T v).
  Parameter P2T_is_P :
    ∀ (t : T) (H : is_P t), P2T (T2P t H) = t.
  Parameter proof_irr :
    ∀ x (p1 p2:is_P x), p1 = p2.
End family.
```

We assume that we have the type T and a restricting predicate is_P as well as the type P. We also provide two conversion functions T2P and P2T, together with two proofs: a proof is_P_lemma that is_P holds for all images (P2T v) of the inhabitants v of P, and a proof P2T_is_P that for all inhabitants t : T satisfying the predicate is_P, P2T is a left inverse of T2P.

Then, the roundtrip lemma P2rec_PK stating that P2rec_P is a left inverse for rec_P2P can be proved automatically using the functor equiv_family, reproduced in the following code snippet.

```
Module Type equiv_sig (f:family).
Import f.
Parameter rec_P : Type.
Parameter rec_P2P : rec_P → P.
Parameter P2rec_P : P → rec_P.
Parameter P2rec_PK : ∀ x: rec_P, P2rec_P (rec_P2P x) = x.
End equiv_sig.

Module equiv_family (Import f:family) <: equiv_sig(f).
  Record rrec_P := Build_rrec_P {
    P_struct :> T;
    P_prop : is_P P_struct
  }.

  Definition rec_P := rrec_P.

  Definition rec_P2P m := T2P (P_struct m) (P_prop m).
  Definition P2rec_P (x:P) : rec_P := Build_rrec_P (P2T x) (is_P_lemma x).
  Lemma P2rec_PK : ∀ x: rec_P, P2rec_P (rec_P2P x) = x.
  Proof.
    unfold rec_P2P, P2rec_P; intros; simpl.
    generalize (is_P_lemma (T2P (P_struct x) (P_prop x))).
    rewrite P2T_is_P.
    intros; destruct x; simpl in *.
    rewrite (proof_irr _ P_prop0 i).
    reflexivity.
  Qed.
End equiv_family.
```

The proof of `P2rec_PK` is generic and only relies on the components of the module `f` which has type `family`.

The proof of the other roundtrip lemma `rec_P2PK` cannot be derived abstractly using a functor. Indeed, the argument of this lemma is an element `m` of the inductively-defined type `P`. Therefore no proof can be carried out before we have an explicit definition of `P`. Once this definition is provided, the proof of the second lemma is rather straightforward and can be automated using some Ltac constructs. Although the Ltac proof is not generic, it works easily for all examples provided in this paper. We believe that this could be generalized to arbitrary datatypes by using some meta-programming tools such as Coq-elpi [10] or MetaCoq [18].

In the next subsection, we shall extend our interface and build a new functor to automatically generate some random generators for the two representations `P` and $\{x : \mathtt{T} \mid \mathtt{is\_P}\ x\}$ at stake.

2.4 Random Generators

Property based testing (PBT) has become famous in the community of functional languages. Mainly popularized by QuickCheck [4] in Haskell, PBT is also available in proof assistants. In Coq, the random testing plugin QuickChick [12] allows us to check the validity of executable conjectures with random inputs, before trying to write formal proofs of these conjectures. QuickChick is mainly a generic framework providing combinators to write testing code, in particular random generators, and also to prove their correctness.

Our general framework also provides guidelines to develop random generators for all the datatypes under study. Generators, either user-defined or automatically derived by QuickChick, have a type `G Ty` where `Ty` is the type of the generated values and `G` is an instance of the Coq `Monad` typeclass. They are usually parameterized by a natural number `n` that controls their termination (called *fuel* in the Coq community). It may also serve as a bound on the depth of the generated values, even if it is not always guaranteed.

Let us assume that a random generator of values of type `T`, named `gen_T : nat → G T`, is available. We are interested in providing a generator for each datatype: (i) a generator of values of type `T` satisfying the property `is_P`, (ii) a generator of values of type `rec_P` embedding a value of type `T` and a proof that the latter satisfies the property `is_P`, (iii) a generator of values of type `P`. Thanks to QuickChick and the bijections we have previously defined, they can be obtained quite easily, using three new functors explained below. All these generators come in a sized version, i.e. they are parameterized with a natural number which is randomly chosen, when used with a QuickChick test command.

The first functor we propose, `generators_family1`, allows the definition of the random generator `gen_filter_P` which implements the strategy *generate and test*. It can be obtained when are available an executable version of the predicate `is_P`, named `is_Pb`, and a proof of decidability of `is_P`, named `is_P_dec`. A value `default_P` of the considered family - which is guaranteed by a proof `default_is_P` - is also required.

```
Module Type family_for_generators1 (Import f : family).
  Import f.
  Module facts := equiv_family (f).
  Parameter is_Pb : T → bool.
  Parameter is_P_dec : ∀ x:T, is_P x ↔ is_Pb x = true.
  Parameter gen_T : nat → G T.
  Parameter default_P : T.
  Parameter default_is_P : is_Pb default_P = true.
End family_for_generators1.

Module generators_family1 (f : family) (g : family_for_generators1 f).
  Import f.
  Import g.
  Import g.facts.

  Definition filter_max := 100.
  Fixpoint gen_filter_P_aux nb n :=
  match nb with
  | 0 ⇒ returnGen default_P
  | S p ⇒ do! val ← gen_T n;
          if is_Pb val then returnGen val
          else gen_filter_P_aux p n
  end.
  Definition gen_filter_P : nat → G T := gen_filter_P_aux filter_max.
End generators_family1.
```

The random generator `gen_filter_P` randomly produces a value `val` of type `T` thanks to `gen_T` and checks whether `is_Pb val` is true, in which case it outputs `val`. Otherwise, it discards the value and tries again. If the maximum number of tries `filter_max` is reached, it yields the provided default value `default_P`.

The next two functors can be used to derive a random generator for one family representation from that of the alternative representation. When the random generator `gen_P` of values of type `P` is available, using the functor `generators_family3` shown below, we can obtain a random generator of values of type `rec_P`, i.e. a value of type `T` and a proof that it satisfies the property `is_P` (thanks to the functions and lemmas derived using `equiv_family`). The functor `generators_family2` (omitted here) does the opposite job.

```
Module Type family_for_generators3 (Import f : family).
  Parameter gen_P : nat → G P.
End family_for_generators3.

Module generators_family3 (Import f : family)
 (Import g : family_for_generators3 f)
 (Import facts : equiv_sig f).
  Definition gen_rec_P n : G rec_P :=
  do! p ← gen_P n;
  returnGen (P2rec_P p).
End generators_family3.
```

In the next section, we shall see how to instantiate our framework with two different representations of closable Motzkin trees and uniquely closable Motzkin trees, to automatically prove the equivalence between the representations and to automatically derive random generators.

3 Two Instances: Closable Motzkin Trees and Uniquely Closable Motzkin Trees

This section presents two simple examples of infinite families of objects with two representations in Coq. These examples are presented as applications of our formal framework, including formal proofs of isomorphisms between representations and the design of their corresponding random generators. Whereas our methodology applies to any pair of isomorphic datatypes, we have chosen to focus our applications primarily on data families related to the λ-terms from the pure (i.e., untyped) λ-calculus.

Let us briefly recall that the λ-calculus is a universal formalism to represent computations with functions. A *(pure) λ-term* is either a variable (x, y, ...), an *abstraction* $\lambda x.t$, that *binds* the variable x in the λ-term t, or a term of the form $t\,u$ for two λ-terms t and u. The term $\lambda x.t$ represents a function of the variable x. The term $t\,u$ represents an *application* of the function (represented by) t to the function (represented by) u. A variable x in *free* in the term t if it is not bound in t (by some λx). A *closed* term is a term without free variables. Terms are considered up to renaming of their bound variables.

The two examples come from a study for the efficient enumeration of closed λ-terms, by Bodini and Tarau [3], that starts from binary-unary trees, a.k.a. Motzkin trees, that can be seen as skeletons of λ-terms. For self-containment, all the definitions and properties of this study that are formalized here are kindly reminded to the reader.

A *Motzkin tree* is a rooted ordered tree built from binary, unary and leaf nodes. Thus the set of Motzkin trees can be seen as the free algebra generated by the constructors `v`, `l` and `a` of respective arity 0, 1 and 2. Their type in Coq, named `motzkin`, is the following inductive type.

```
Inductive motzkin : Set :=
| v : motzkin
| l : motzkin → motzkin
| a : motzkin → motzkin → motzkin.
```

3.1 Closable Motzkin Trees

The *skeleton* of the λ-term t is the Motzkin tree obtained by erasing all the occurrences of the variables in t. A Motzkin tree is *closable* if it is the skeleton of at least one closed λ-term. As in [3], we define a predicate for characterizing closable Motzkin trees:

```
Fixpoint is_closable (mt: motzkin) :=
  match mt with
  | v ⇒ False
  | l m ⇒ True
  | a m1 m2 ⇒ is_closable m1 ∧ is_closable m2
  end.
```

This predicate only requires the presence of at least one occurrence of the unary node on each rooted path of the Motzkin tree. For instance, the tree `l (a v (l v))` is closable (it is the skeleton of the closed λ-term $\lambda x.x(\lambda y.y)$), whereas the tree `a (l v) v` is not closable.

Bodini and Tarau proposed a grammar generating closable Motzkin trees [3, Section 3], that we adapt in Coq as an inductive type, named `closable`.

```
Inductive closable :=
| La : motzkin → closable
| Ap : closable → closable → closable.
```

Table 1 Two instances of the Module Type family and the functor equiv_family representing closable Motzkin trees and uniquely closable Motzkin trees. Statements required in the functor Module Type family (upper part of the array) are proven automatically. The roundtrip statement rec_P2PK (last line of the array), which corresponds to rec_closable2closableK and rec_ucs2ucsK does not belong to the functor but can be proven automatically in both settings.

Abstraction	Closable Skeletons	Uniquely Closable Skeletons
T	motzkin	motzkin
is_P	is_closable	is_ucs
P	closable	ucs
T2P	motzkin2closable	motzkin2ucs
P2T	closable2motzkin	ucs2motzkin
is_P_lemma	automatically proved using Ltac	
P2T_is_P	automatically proved using Ltac	
proof_irr	proof_irr_is_closable	proof_irr_is_ucs
rec_P	automatically derived in the functor	
rec_P2P	automatically derived in the functor	
P2rec_P	automatically derived in the functor	
P2rec_PK	automatically derived in the functor	
rec_P2PK	automatically proved using Ltac	

For example, La (a v (l v)) is the closable term corresponding to the Motzkin tree l (a v (l v)).

To prove that there is a bijection between closable Motzkin trees specified using the type rec_closable and inductive objects whose type is closable, using our approach, we simply need to provide two functions motzkin2closable and closable2motzkin.

```
Fixpoint motzkin2closable (m : motzkin) : is_closable m → closable :=
  match m as m0 return (is_closable m0 → closable) with
  | v ⇒ fun H : is_closable v ⇒ let H0 := match H return closable with end in H0
  | l m0 ⇒ fun _ : is_closable (l m0) ⇒ La m0
  | a m1 m2 ⇒ fun H : is_closable (a m1 m2) ⇒
      match H with
      | conj Hm1 Hm2 ⇒ Ap (motzkin2closable m1 Hm1) (motzkin2closable m2 Hm2)
      end
  end.
```

```
Fixpoint closable2motzkin c :=
  match c with
  | La m ⇒ l m
  | Ap c1 c2 ⇒ a (closable2motzkin c1) (closable2motzkin c2)
  end.
```

Because it involves dependent pattern matching, defining directly motzkin2closable as a function is not immediate. However it is easily carried out interactively as a lemma, in a proof-like manner, using the tactic fix.

The transformation functions and the isomorphism properties between the two types closable and rec_closable can then be automatically generated, as summarized in the second column of Table 1.

3.1.1 Random Generators

Random generators for closable and rec_closable have been used to test the different lemmas before proving them, for example the roundtrip lemma rec_closable2closableK, which is an instance of the pattern rec_P2PK. Corresponding QuickChick commands can be found in our formal development.

The generator for Motzkin trees, gen_motzkin, required by any of the other generators, is obtained automatically, thanks to QuickChick:

```
Derive (Arbitrary, Show) for motzkin.
```

In the context of closable Motzkin trees, the gen_closable generator associated to the tailored simple inductive type closable can be easily obtained using QuickChick. Thanks to the functor generators_family3, we can derive the random generator of values of the corresponding restricted type, as it is illustrated by the following snippet of code, where closable is an instance of the family, and fact_cl is defined as the module equiv_family (closable).

```
Module gen_closable3 : family_for_generators3 (closable).
Definition gen_P := gen_closable.
End gen_closable3.

Module V3 := generators_family3 closable gen_closable3 facts_cl.
```

To test the motzkin2closable function (T2P in the family interface), we need a generator that produces closable Motzkin trees. It is not relevant to use the previously defined generator which we have derived from that of closable values and thus obtained using, as a main ingredient, the function under test itself. For that purpose, the generator gen_filter_P obtained by applying the functor generators_family1 can be useful, however such a generator usually discards many values to produce the required ones. A handmade generator, as gen_closable_struct defined below, is usually preferred.

As a representative of this kind of custom generators, we expose its code in the following code snippet and explain it.

```
Fixpoint gen_closable_struct_aux (k : nat) (n : nat) : G motzkin :=
match n with
| 0 ⇒ match k with
      0 ⇒ returnGen default_closable
      | _ ⇒ returnGen v
      end
| S p ⇒
  match k with
  0 ⇒ oneOf [
    (returnGen default_closable);
    (do! mt ← gen_closable_struct_aux (S k) p; returnGen (l mt));
    (do! mt0 ← gen_closable_struct_aux k p; do! mt1 ← gen_closable_struct_aux k p;
      returnGen(a mt0 mt1)) ]
  | _ ⇒ oneOf [
    (returnGen v);
    (do! mt ← gen_closable_struct_aux (S k) p; returnGen (l mt));
    (do! mt0 ← gen_closable_struct_aux k p; do! mt1 ← gen_closable_struct_aux k p;
      returnGen(a mt0 mt1)) ]
  end
end.

Definition gen_closable_struct : nat → G motzkin := gen_closable_struct_aux 0.
```

We first define an intermediate function that uses the additional parameter `k` denoting the number of `l` constructors at hand. So, if both `k` and `n` are equal to 0, the generator emits the default value (here `l v`, stored in `default_closable`). If `n` is 0 but at least one `l` is available, then the generator produces the leaf `v`. When `n` is not 0, again we have two treatments depending on whether we have already introduced the constructor `l` or not. In both cases, the generator picks one of the several ways to produce a value – thanks to `oneOf`, and thus either stops with a value (resp. `l v` or `v`), recursively produces a closable Motzkin tree which is used to build a resulting unary Motzkin tree, or recursively generates two closable Motzkin trees used to produce a binary Motzkin tree. The final custom generator is obtained using the previous intermediate function with `k` equal to 0.

We recommend testing that this generator does produce Motzkin trees which are closable, as follows:

```
QuickCheck (sized (fun n ⇒ forAll (gen_closable_struct n) is_closableb)).
(*  +++ Passed 10000 tests (0 discards) *)
```

To define the proof-carrying version of the custom generator, we follow a similar scheme but also produce a proof that the produced value `mt` is closable, i.e. a term of type `is_closable mt`. We use the `Program` facility which allows us to produce certified programs and generates proof obligations. Here these proof obligations are automatically solved.

3.2 Uniquely Closable Motzkin Trees

A Motzkin tree is *uniquely closable* if there exists exactly one closed λ-term having it as its skeleton.

We first define a predicate `is_ucs` for characterizing uniquely closable skeletons. This predicate specifies that a Motzkin tree is uniquely closable if and only if there is exactly one unary node on each rooted path.

```
Fixpoint is_ucs_aux m b :=
  match m with
  | v ⇒ b = true
  | l m ⇒ if b then False
          else is_ucs_aux m true
  | a m1 m2 ⇒ is_ucs_aux m1 b ∧ is_ucs_aux m2 b
  end.

Definition is_ucs m := is_ucs_aux m false.
```

This Coq predicate corresponds to the second Prolog predicate `uniquelyClosable2` introduced by Bodini and Tarau [3, Section 4], after a first Prolog predicate `uniquelyClosable1` using a natural number to count the number of λ binders above each leaf, instead of a Boolean flag as here. A Coq formalization of this other characterization of uniquely closable Motzkin trees, and a formal proof of their equivalence, are presented in Section 5.4.

We then define an inductive type `ucs` that also represents uniquely closable Motzkin trees.

```
Inductive ca :=
| V : ca
| B : ca → ca → ca.

Inductive ucs :=
| L : ca → ucs
| A : ucs → ucs → ucs.
```

Even though we use the abbreviations `ca` for `ClosedAbove` and `ucs` for `UniquelyClosable`, these types exactly correspond to Haskell datatypes given in [3]. For instance, the Motzkin tree `l (a v v)` and the corresponding `ucs` term `L (B V V)` represent uniquely closable skeletons. The closable tree `l (a (l v) v)` is not uniquely closable, because it is the skeleton of two closed λ-terms, namely $\lambda x.(\lambda y.y)x$ and $\lambda x.(\lambda y.x)x$.

Using the same infrastructure as for closable Motzkin trees, the transformation functions and the isomorphism properties between the two types `ucs` and `rec_ucs` can be automatically generated, as summarized in the last column of Table 1.

We proceed in the same way for random generators. Using QuickChick, the generator `gen_ucs` is automatically derived from the definition of the inductive types `ca` and `ucs`. The user-defined generator `gen_ucs_struct` is very close to `gen_closable_struct`. Similarly we use `Program` to define the one producing values and proofs.

4 Pure Open λ-Terms in De Bruijn Form

Let us now address the questions of formal representations and random generation of pure open λ-terms modulo variable renaming. The definitions in this section are not present in Bodini and Tarau's work [3].

To get rid of variable names, we adopt de Bruijn's proposal to replace each variable in a λ-term by a natural number, called its *de Bruijn index* [8]. When a de Bruijn index is not too high, it encodes a variable bound by the number of λ's between its location and the λ that binds it. Otherwise, it encodes a free variable. We consider de Bruijn indices from 0, to ease their formalization with the Coq type `nat` for natural numbers. For instance, the term $\lambda.(1\ (\lambda.1))$ in de Bruijn form represents the term $\lambda x.(y\ (\lambda z.x))$ with the free variable y.

4.1 Types

The tree structure of open λ-terms in de Bruijn form can be represented by unary-binary trees whose leaves are labeled by a natural number. They are the inhabitants of the following inductive Coq type `lmt` (acronym for `l`abeled `M`otzkin `t`ree).

```
Inductive lmt : Set :=
| var : nat → lmt
| lam : lmt → lmt
| app : lmt → lmt → lmt.
```

However the property of being closed cannot be defined by induction on this definition of λ-terms. Indeed, if the term $\lambda\, t$ is closed, then the term t is not necessarily closed, it can also have a free variable. The more general property of m-openness overcomes this limitation: for any natural number m, the λ-term t is said to be *m-open* if the term $\lambda \ldots \lambda\, t$ with m abstractions before t is closed. Whereas the "m-open" terminology is recent [1], the notion has been studied since 2013, by Grygiel and Lescanne [11, 13].

With the following definition, `(is_open m t)` holds iff the labeled Motzkin tree `t` encodes an `m`-open λ-term. This function call indeed visits the tree `t` and counts (from `m`) the number of λs (constructor `lam`) traversed so far. At each leaf (constructor `var`) it checks that its de Bruijn indice `i` is lower than this number `m` of traversed abstractions.

```
Fixpoint is_open (m: nat) (t: lmt) : Prop :=
  match t with
  | var i ⇒ i < m
  | lam t1 ⇒ is_open (S m) t1
  | app t1 t2 ⇒ is_open m t1 ∧ is_open m t2
  end.
```

For instance, the tree `lam (app (var 0) (lam (var 1)))` is 0-open (its skeleton is the closable term `l (a v (l v))`), whereas the tree `lam (app (var 1) (lam (var 1)))` is 1-open, but not 0-open.

Because of the extra parameter m, the formal framework presented in Sect. 2 must be adapted and we propose a new module type `param_family` together with a functor `equiv_param_family` to automatically prove one of the roundtrip lemmas. The other one can be easily proved correct using the same sequences of Ltac constructs as for the non dependent case.

The following record type parameterized by `m` is such that `(rec_open m)` describes `m`-open terms. As previously, the first field stores the datum, here a labeled Motzkin tree (i.e., `T` is `lmt`), and the second field stores a proof that it is `m`-open.

```
Record rec_open (m:nat) : Set := Build_rec_open {
  open_struct :> lmt;
  open_prop : is_open m open_struct
}.
```

It is however more natural to describe `m`-open terms with a dependent type `(open m)` enclosing the condition `i < m` at leaves, as follows.

```
Inductive open : nat → Set :=
| open_var : ∀ (m i:nat), i < m → open m
| open_lam : ∀ (m:nat), open (S m) → open m
| open_app : ∀ (m:nat), open m → open m → open m.
```

4.2 Transformations and Their Properties

In order to switch from one representation to the other whenever needed, we provide two functions `rec_open2open m` and `open2rec_open m`, and Coq proofs for two roundtrip lemmas justifying that they are mutual inverses.

From the Record Type to the Dependent Type. The function `rec_open2open m` from the record type `(rec_open m)` to the dependent type `(open m)` is defined by

```
Definition rec_open2open (m : nat) (r : rec_open m) :=
  lmt2open (open_struct m r) m (open_prop m r).
```

where `lmt2open` is the following dependent recursive function.

```
Fixpoint lmt2open (t:lmt) : ∀ m:nat, is_open m t → open m :=
  match t as u return (∀  m0 : nat, is_open m0 u → open m0) with
  | var n ⇒ fun (m0 : nat) (H : is_open m0 (var n)) ⇒ open_var m0 n H
  | lam u ⇒ fun (m0 : nat) (H : is_open m0 (lam u)) ⇒
      open_lam m0 (lmt2open u (S m0) H)
  | app u w ⇒
      fun (m0 : nat) (H : is_open m0 (app u w)) ⇒
        match H with
        | conj H0 H1 ⇒ open_app m0 (lmt2open u m0 H0) (lmt2open w m0 H1)
        end
end.
```

It is rather difficult to define this function directly. We choose to develop it as a proof, as advocated by McBride [15], in an interactive manner, letting Coq handle the type dependencies. Once the term is built, we simply revert the proof and declare it directly as a fixpoint construction to make it look like a function, more readable and understandable for humans than a proof script.

From the Dependent Type to the Record Type. The process to define the inverse function `open2rec_open m` from the dependent type (`open m`) to the record type (`rec_open m`) is rather different, and can be decomposed as follows. First of all, a function (`open2lmt m`) turns each dependent term `t` of type `open m` into a labeled Motzkin tree.

```
Fixpoint open2lmt (m:nat) (t : open m) : lmt :=
  match t with
  | open_var m i _ ⇒ var i
  | open_lam m u ⇒ lam (open2lmt (S m) u)
  | open_app m t1 t2 ⇒ app (open2lmt m t1) (open2lmt m t2)
  end.
```

Then we prove automatically, using the same Ltac constructs as for the previous examples, the following lemma that states that the function `open2lmt m` always outputs an `m`-open term.

```
Lemma is_open_lemma : ∀ m t, is_open m (open2lmt m t).
```

Once this lemma is proved, we can derive automatically the transformation `open2rec_open`, by using the functor `equiv_param_family`.

```
Definition open2rec_open m t := Build_rec_open m (open2lmt m t) (is_open_lemma m t).
```

As we did in the previous sections, we then need to prove a lemma `open2lmt_is_open` which relates the functions `open2lmt` and `lmt2open`, without taking into account the restriction property.

```
Lemma open2lmt_is_open : ∀ m t H, open2lmt m (lmt2open t m H) = t.
```

Both lemmas are part of the interface `param_family` for a parametric family, extending the interface `family`. Thus, applying the appropriate functor, we automatically derive a proof of the first roundtrip lemma:

```
Lemma open2rec_openK : ∀ m r, open2rec_open m (rec_open2open m r) = r.
```

The proof of the second roundtrip lemma proceeds by induction on `x` of type `open m`. It is immediately proven using the Ltac constructs proposed in the previous sections.

```
Lemma rec_open2openK : ∀ m x, rec_open2open m (open2rec_open m x) = x.
```

4.3 Random Generators

The required generator `gen_lmt` is automatically derived by QuickChick from the definition of the inductive type `lmt`. The custom generators for λ-terms satisfying the `open m` property, with or without proofs, are written following the same canvas as before. The generator corresponding to the inductive type `open` is no longer derived automatically by QuickChick, in particular because proofs have to be inserted when using the `open_var` constructor. However it is easy to define it manually.

4.4 Characterization of Open λ-Terms From Their Skeleton

This subsection presents definitions and formal proofs relating Bodini and Tarau's skeletons for λ-terms (Section 3) with m-open λ-terms introduced in this section, not present in Bodini and Tarau's work.

The skeleton of a λ-term is the Motzkin tree obtained by erasing the labels at its leaves.

```
Fixpoint skeleton (t: lmt) : motzkin :=
  match t with
  | var _ ⇒ v
  | lam t1 ⇒ l (skeleton t1)
  | app t1 t2 ⇒ a (skeleton t1) (skeleton t2)
  end.
```

This function (specified by `toMotSkel` in [3]) connects Motzkin trees without labels (Sect. 3) and Motzkin trees with labels defined in this section.

As the `skeleton` function cannot be inverted functionality, we define a pseudo-reverse, from Motzkin trees without labels to labeled Motzkin trees, as the following family of inductive relations (`label m`), for all natural numbers `m`.

```
Inductive label : nat → motzkin → lmt → Prop :=
| Lvar : ∀ m i, i < m → label m v (var i)
| Llam : ∀ m mt t, label (S m) mt t → label m (l mt) (lam t)
| Lapp : ∀ m mt1 mt2 t1 t2, label m mt1 t1 → label m mt2 t2
    → label m (a mt1 mt2) (app t1 t2).
```

The label-removing function `skeleton` and the label-adding relation `label` can be used together as follows, to define a second characterization of `m`-open λ terms among labeled Motzkin trees `t`.

```
Definition skeleton_open (m:nat) (t:lmt) : Prop := label m (skeleton t) t.
```

The proof of the following equivalence with the first characterization (`is_open`, introduced in Section 4) is straightforward.

```
Lemma skeleton_is_open_eq : ∀ m t, skeleton_open m t ↔ is_open m t.
```

An m_1-open λ-term is also an m_2-open λ-term for all $m_2 \geq m_1$.

```
Lemma label_mon : ∀ m1 mt t, label m1 mt t → ∀ m2, m1 ≤ m2 → label m2 mt t.
```

Consequently, for any labeled Motzkin tree t, there is a minimal natural number m such that t is an m-open λ-term. It can be computed for instance by the following function.

```
Fixpoint minimal_openness (t : lmt) : nat :=
  match t with
  | var i ⇒ i+1
  | lam t ⇒ match minimal_openness t with S m ⇒ m | _ ⇒ 0 end
  | app t1 t2 ⇒ max (minimal_openness t1) (minimal_openness t2)
  end.
```

The function `skeleton` and the relation `label` are pseudo-inverses in the sense of the following two lemmas.

```
Lemma label_skeletonK : ∀ t : lmt, label (minimal_openness t) (skeleton t) t.
```

```
Lemma skeleton_labelK : ∀ m : nat, ∀ mt : motzkin, ∀ t : lmt,
  label m mt t → skeleton t = mt.
```

The lemmas `label_skeletonK` and `skeleton_is_open_eq` jointly establish that the labeled Motzkin tree `t` is a (`minimal_openness t`)-open λ-term.

```
Lemma lmt_minimal_openness : ∀ t : lmt, is_open (minimal_openness t) t.
```

Finally, it is easy to prove by induction that `minimal_openness t` indeed computes the smallest openness m such that `t` is an m-open λ-term.

```
Lemma minimality : ∀ t : lmt, ∀ m : nat, is_open m t → m ≥ minimal_openness t.
```

5 Use Cases

In this section we use the previous examples of types to formalize all the propositions in Bodini and Tarau's work [3] that are related to Motzkin trees and pure λ-terms.

5.1 Another Definition for Closable Skeletons

Bodini and Tarau [3, section 3] first defined closable skeletons with a Prolog predicate – named `isClosable` – whose adaptation in Coq is

```
Fixpoint isClosable2 (mt: motzkin) (V: nat) :=
  match mt with
  | v ⇒ V > 0
  | l m ⇒ isClosable2 m (S V)
  | a m1 m2 ⇒ isClosable2 m1 V ∧ isClosable2 m2 V
  end.

Definition isClosable (mt: motzkin) := isClosable2 mt 0.
```

For each λ binder this function increments a counter `V` (starting at `0`). Then it checks at each leaf that its label is strictly positive. This definition is slightly more complicated than that of the Coq predicate `is_closable` presented in Sect. 3. We have proved formally that both definitions are equivalent:

```
Lemma is_closable_isClosable_eq : ∀ (mt: motzkin), is_closable mt ↔ isClosable mt.
```

The two implications of this equivalence are proved by structural induction and thanks to the following two lemmas, themselves proved by structural induction.

```
Lemma isClosable2_S : ∀ m n, isClosable2 m n → isClosable2 m (S n).
Lemma isClosable_l : ∀ m, isClosable (l m).
```

We can notice that this proof is simpler than expected: Although the generalization `isClosable2` is required to define the predicate `isClosable`, the proof avoids the effort to invent generalizations to `isClosable2` of the predicate `is_closable` and the equivalence lemma. Similarly, after "packing" the predicate `isClosable` in the following record type, it was possible to define and prove isomorphism with the algebraic datatype `closable` without having to generalize the record and the datatype to `isClosable2`.

```
Record recClosable : Type := Build_recClosable {
  Closable_struct : motzkin;
  Closable_prop : isClosable Closable_struct
}.
```

5.2 Two Definitions for the Size of Terms

Bodini and Tarau [3, Proposition 1] state the following proposition to justify that two different size definitions lead to the same sequence of numbers of closed λ-terms modulo variable renaming, counted by increasing size.

▶ **Proposition 1.** *The set of terms of size n for size defined by the respective weights 0, 1 and 2 for variables, abstractions and applications is equal to the set of terms of size $n+1$ for size defined by weight 1 for variables, abstractions and applications.*

This proposition holds not only for all Motzkin trees (without labels), but also for closable ones, labeled ones, and for m-open λ-terms. Since we proposed two Coq types for closable Motzkin trees and for m-open λ-terms, we formalize Proposition 1 by six propositions in Coq, all of the form

```
Proposition proposition1X : ∀ t : X, size111X t = size012X t + 1.
```

with `X` in {`motzkin`, `rec_closable`, `closable`, `lmt`, `rec_open`, `open`}, and with adequate functions `size111X` and `size012X`, not detailed here, defining both sizes for each type. More precisely, thanks to the coercion (`P_struct :> T`) in the record types, the functions `size*rec_P` are not defined, but advantageously replaced by the functions `size*T`. Here, `*` is either `111` or `012` and (`T`,`P`) is either (`motzkin`,`closable`) or (`lmt`,`open`). For record types, the proposition then takes the following form:

```
Proposition proposition1rec_P : ∀ t : rec_P, size111T t = size012T t + 1.
```

This proposition is a straightforward consequence of the corresponding proposition on the type `T` (named `proposition1T`, according to our naming conventions). This mechanism being similar for all record types, it can easily be mechanized.

The situation is very different with – potentially – dependent types (named `P` in our general framework), if we forbid ourselves to use their isomorphism with a record type to prove their proposition (named `proposition1P`, according to our naming conventions). Here, the propositions for `P` in {`closable`,`open`} are proved by structural induction and linear arithmetic, because the latter suffices to inductively define the size functions. However, the general situation may be arbitrarily more complex, so no general mechanization can be considered.

5.3 Characterization of Closable Motzkin Trees

This section and the next one present two propositions from Bodini and Tarau's work [3] that cannot be formalized with the single unlabeled notion of skeleton introduced in that work, but also require a formalization of λ-terms with labels for their variables, such that the one introduced in Section 4 of the present paper.

The first of these two propositions is the following characteristic property for closable Motzkin trees [3, Proposition 2].

▶ **Proposition 2.** *A Motzkin tree is the skeleton of a closed λ-term if and only if it exists at least one λ-binder on each path from the leaf to the root.*

After defining a closed λ-term as a 0-open λ-term, we can state Proposition 2 in Coq, as follows.

```
Definition is_closed t := is_open 0 t.
Proposition proposition2 : ∀ mt : motzkin,
  (∃ t : lmt, skeleton t = mt ∧ is_closed t) ↔ is_closable mt.
```

This formalization is close to the text of Proposition 2. It relies on the base type `motzkin` and the restriction `is_closable`. A formulation of this proposition with the type `rec_closable` or `closable` would be useless, because these more precise types already include the *closability* property characterized by the proposition.

The proof of this proposition is straightforward.

5.4 Characterization of Uniquely Closable Motzkin Trees

Bodini and Tarau propose the following characteristic property for uniquely closable Motzkin trees [3, Proposition 4].

▶ **Proposition 3.** *A skeleton is uniquely closable if and only if exactly one lambda binder is available above each of its leaf nodes.*

The predicates `is_ucs` and `is_ucs_aux` presented in Section 3.2 correspond to the Prolog predicate `uniquelyClosable2` of [3, Section 4] and to the characteristic property "exactly one lambda binder is available above each of its leaf nodes" of Proposition 4 of [3]. Therefore, proving Proposition 3 consists in showing that this property is equivalent to the definition "We call a skeleton *uniquely closable* if it exists exactly one closed lambda term having it as its skeleton." [3, page 6], which gives the following Coq code.

```
Proposition proposition4: ∀ mt : motzkin,
  (∃! t, skeleton t = mt ∧ is_closed t) ↔ is_ucs mt.
```

However, this proposition cannot be proved directly, because `(is_closed t)` is a special case of `(is_open m t)`, which is parametrized by a natural number `m`, while `(is_ucs mt)` is a special case of of `(is_ucs_aux mt b)`, which is only parameterized by a Boolean `b`. The rest of this section addresses this issue by generalizing the proposition to any natural number `m`, using a characterization `(ucs1_aux mt m)` parametrized by this integer and put in correspondence with `(is_ucs_aux mt b)`.

The following predicates `ucs1_aux` and `ucs1` adapt in Coq the Prolog predicate named `uniquelyClosable1` in [3].

```
Fixpoint ucs1_aux (t:motzkin) (n:nat) : Prop :=
  match t with
  | v ⇒ (1 = n)
  | l m ⇒ ucs1_aux m (S n)
  | a m1 m2 ⇒ ucs1_aux m1 n ∧ ucs1_aux m2 n
  end.
Definition ucs1 (t:motzkin) := ucs1_aux t 0.
```

We then use the predicate `ucs1_aux` to state a generalization of Proposition 3 to any openness m, then the predicate `ucs1` to state its specialization when $m = 0$, which is a variant of Proposition 3.

```
Lemma proposition4ucs1_aux : ∀ (mt : motzkin) (m : nat),
  (∃! t, skeleton t = mt ∧ is_open m t) ↔ ucs1_aux mt m.
```

```
Corollary proposition4ucs1: ∀ mt : motzkin,
  (∃! t, skeleton t = mt ∧ is_closed t) ↔ ucs1 mt.
```

Independently, we can prove that the two charaterizations of uniquely closable Motzkin trees are equivalent.

```
Lemma ucs1_is_ucs_eq : ∀ mt : motzkin, ucs1 mt ↔ is_ucs mt.
```

As usually when formalizing pen-and-paper proofs, we get more precise statements and more detailed proofs. For example, we formally proved Proposition 1 in [3] as four propositions, corresponding to four distinct data families.

6 Conclusions and Perspectives

We have presented a framework to define and formally prove isomorphisms between Coq datatypes, and to produce random generators for them. After applying it to several examples related to lambda term families, we have formalized in Coq a large subset of the computational and logical content of Bodini and Tarau's paper [3] about pure λ-terms. Although our work is clearly dealing with Coq representations, our technique could be useful to other proof assistant tools and could be developed for example in Isabelle/HOL or Agda.

Technically, our present approach using interfaces allows us to automatically derive only one of two round-trip properties, that state that the considered transformations are inverse bijections. The other one, which proceeds by induction on the type `P`, cannot be generated automatically by a functor, however, we can prove it automatically using some advanced tactic combinations using Ltac. In the near future, we plan to investigate in more details whether using external tools like MetaCoq [18] or elpi [10] and Coq-elpi [10] would increase the genericity of our approach compared to simply relying on Ltac.

Our framework obviously applies to other formalization topics. It was inspired by previous work, including one on Coq representations of permutations and combinatorial maps [9]. We plan to complete this work and revisit it using this structuring framework. The proofs of isomorphisms presented in this paper were elementary because the two types in bijection were very close to one another. In the more general case of two different points of view on the same family (e.g., permutations seen as injective endofunctions or products of disjoint cycles), isomorphisms can be arbitrarily more difficult to prove.

References

1 Maciej Bendkowski, Olivier Bodini, and Sergey Dovgal. Statistical Properties of Lambda Terms. *The Electronic Journal of Combinatorics*, 26(4):P4.1, October 2019. `doi:10.37236/8491`.

2 Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development, Coq'Art : The Calculus of Inductive Constructions*. Springer-Verlag, Berlin/Heidelberg, May 2004. 469 pages.

3 Olivier Bodini and Paul Tarau. On uniquely closable and uniquely typable skeletons of lambda terms. In Fabio Fioravanti and John P. Gallagher, editors, *Logic-Based Program Synthesis and Transformation. LOPSTR 2017*, volume 10855 of *Lecture Notes in Computer Science*, pages 252–268. Springer, Cham, 2018. `doi:10.1007/978-3-319-94460-9_15`.

4 Koen Claessen and John Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*, volume 35 of *SIGPLAN Not.*, pages 268–279. ACM, New York, 2000. `doi:10.1145/351240.351266`.

5 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical type theory: A constructive interpretation of the univalence axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

6 Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs. CPP 2013*, volume 8307 of *Lecture Notes in Computer Science*, pages 147–162. Springer, Cham, 2013. `doi:10.1007/978-3-319-03545-1_10`.

7 Coq development team. *The Coq Proof Assistant Reference Manual, Version 8.13.2*. INRIA, 2021. URL: `http://coq.inria.fr`.

8 N. G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972. doi:10.1016/1385-7258(72)90034-0.

9 Catherine Dubois and Alain Giorgetti. Tests and proofs for custom data generators. *Formal Aspects of Computing*, 30(6):659–684, July 2018. doi:10.1007/s00165-018-0459-1.

10 Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: Fast, Embeddable, λProlog Interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning. LPAR 2015*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, Berlin, Heidelberg, 2015. doi:10.1007/978-3-662-48899-7_32.

11 Katarzyna Grygiel and Pierre Lescanne. Counting and generating lambda terms. *Journal of Functional Programming*, 23(5):594–628, September 2013. doi:10.1017/S0956796813000178.

12 Leonidas Lampropoulos and Benjamin C. Pierce. *QuickChick: Property-Based Testing in Coq*. Software Foundations series, volume 4. Electronic textbook, August 2022. Version 1.3.1 https://softwarefoundations.cis.upenn.edu/qc-1.3.1.

13 Pierre Lescanne. On counting untyped lambda terms. *Theoretical Computer Science*, 474:80–97, February 2013. doi:10.1016/j.tcs.2012.11.019.

14 Nicolas Magaud. Changing data representation within the Coq system. In David A. Basin and Burkhart Wolff, editors, *Theorem Proving in Higher Order Logics. TPHOLs 2003*, volume 2758 of *Lecture Notes in Computer Science*, pages 87–102. Springer, Berlin, Heidelberg, 2003. doi:10.1007/10930755_6.

15 Conor McBride. Elimination with a motive. In Paul Callaghan, Zhaohui Luo, James McKinna, and Robert Pollack, editors, *Types for Proofs and Programs. TYPES 2000*, volume 2277 of *Lecture Notes in Computer Science*, pages 197–216. Springer, Berlin, Heidelberg, 2000. doi:10.1007/3-540-45842-5_13.

16 Conor McBride and James McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004. doi:10.1017/S0956796803004829.

17 Zoe Paraskevopoulou, Aaron Eline, and Leonidas Lampropoulos. Computing correctly with inductive relations. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, PLDI 2022, pages 966–980. ACM, New York, 2022. doi:10.1145/3519939.3523707.

18 Matthieu Sozeau, Abhishek Anand, Simon Boulier, Cyril Cohen, Yannick Forster, Fabian Kunze, Gregory Malecha, Nicolas Tabareau, and Théo Winterhalter. The MetaCoq project. *Journal of Automated Reasoning*, 64(5):947–999, 2020. doi:10.1007/s10817-019-09540-0.

19 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

20 Philip Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313. ACM Press, 1987. doi:10.1145/41625.41653.

A Semantics of $\mathbb{K}$ into Dedukti

Amélie Ledein
Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Valentin Blot
Université Paris-Saclay, Inria, CNRS, ENS Paris-Saclay, Laboratoire Méthodes Formelles, France

Catherine Dubois
ENSIIE, Samovar, Évry-Courcouronnes, France

Abstract

$\mathbb{K}$ is a semantical framework for formally describing the semantics of programming languages thanks to a BNF grammar and rewriting rules on configurations. It is also an environment that offers various tools to help programming with the languages specified in the formalism. For example, it is possible to execute programs thanks to the generated interpreter, or to check their properties thanks to the provided automatic theorem prover called the KProver. $\mathbb{K}$ is based on Matching Logic, a first-order logic with an application and fixed-point operators, extended with symbols to encode equality, typing and rewriting. This specific Matching Logic theory is called Kore.

Dedukti is a logical framework having for main goal the interoperability of proofs between different formal proof tools. Several translators to Dedukti exist or are under development, in order to automatically translate formalizations written, for instance, in Coq or PVS. Dedukti is based on the $\lambda\Pi$-calculus modulo theory, a λ-calculus with dependent types and extended with a primitive notion of computation defined by rewriting rules. The flexibility of this logical framework allows to encode many theories ranging from first-order logic to the Calculus of Constructions.

In this article, we present a paper formalization of the translation from $\mathbb{K}$ into Kore, and a paper formalization and an automatic translation tool, called KaMeLo, from Kore to Dedukti in order to execute programs in Dedukti.

2012 ACM Subject Classification Theory of computation → Operational semantics

Keywords and phrases Programming language, Semantics, Rewriting, Logical framework, Type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.12

Supplementary Material *Software*: `https://gitlab.com/semantiko/kamelo`
archived at `swh:1:dir:9353026d393b52e5bc149a469a3d2386fb923dff`

Funding *Amélie Ledein*: Digicosme and EuroProofNet

Acknowledgements We want to thank the $\mathbb{K}$ team, especially Andrei Arusoaie, Xiaohong Chen, Denisa Diaconescu, Everett Hildenbrandt, Zhengyao Lin, Dorel Lucanu, Ana Pantilie and Traian-Florin Serbanuta for their prompt responses to our many questions.

1 Introduction

The main objective of formal methods is to obtain greater confidence in programs. Before verifying a program, it must be written in a programming language whose syntax and semantics are precisely known. Therefore, we must first have a formalization of the semantics of the programming language used to write the program we wish to verify. Several tools make it possible to write formal semantics for example Centaur [7], ASF+SDF [23], Ott [20], Sail [3], Lem [17] or $\mathbb{K}$ [19, 2]. In this article, we are only interested in the latter, since there are currently a large number of programming language semantics written in $\mathbb{K}$ such as Java [6], C [13] or JavaScript [18].

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 12; pp. 12:1–12:22
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

$\mathbb{K}$ is a semantical framework that offers many features for writing a semantics such as attributes to specify evaluation strategies. Once the semantics of a language $\mathcal{L}$ has been specified, $\mathbb{K}$ allows to execute a program $\mathcal{P}$ written in $\mathcal{L}$ but also offers the possibility of verifying some properties – expressed in the form of reachability properties – on the program $\mathcal{P}$ using the automatic theorem prover KProver [21]. As it is possible to automatically translate $\mathbb{K}$ semantics into a Matching Logic theory named Kore, the particularity of the $\mathbb{K}$ framework is to see any semantics $\mathcal{S}$ of a programming language $\mathcal{L}$ as a logical theory $\Gamma^{\mathcal{L}}$. That is the reason we look at the translation of $\mathbb{K}$ into Dedukti, which is a logical framework based on the $\lambda\Pi$-calculus modulo theory having for main goal the interoperability of proofs between different formal proof tools.

In this article, we are more particularly interested in the translation into Dedukti of any semantics written in $\mathbb{K}$ in order to execute programs in Dedukti. Our first contribution is a paper formalization of the translation from $\mathbb{K}$ into Kore. As no article has been yet published on this translation, this contribution was elaborated by reverse engineering on Kore files as well as thanks to discussions with the $\mathbb{K}$ team. Independently, we formalize transformations similar to what was done in IsaK [15, 16], which is a formalization in Isabelle – but not based on Kore – of the static and dynamic semantics of $\mathbb{K}$. In addition, the second contribution is a paper formalization and an automatic tool, called KaMeLo [1], from Kore to Dedukti in order to execute programs in Dedukti. The general overview of the translation pipeline presented in this article is available in Figure 1. This is the first translation into Dedukti involving a semantical framework.

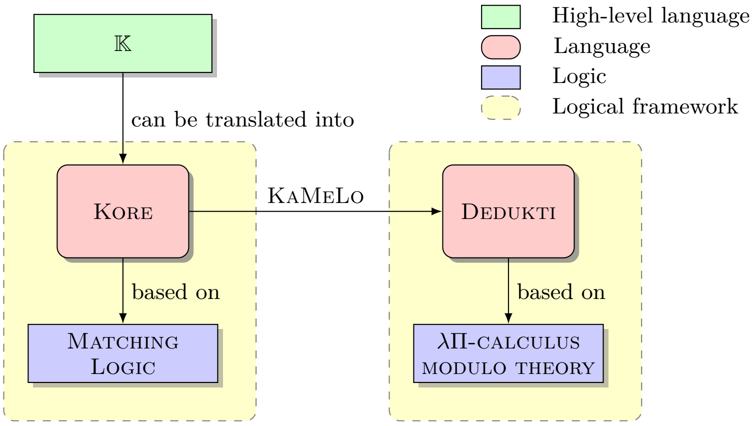


Figure 1 Overview of the translation pipeline presented in this article.

A long-term goal is not only to execute a program $\mathcal{P}$ written with the formalized language $\mathcal{L}$ within Dedukti, but also to verify the proofs established by the KProver, or even make this proof with Dedukti if the KProver has failed, and also formally check meta-properties about the language $\mathcal{L}$. This long-term goal can be seen as a new pipeline for program verification, which is parametrized by a programming language and leaves the user free to choose the proof assistant they wish.

This article is structured as follows: first, we explain how to write a $\mathbb{K}$ semantics (Section 2). Then, we present a mathematical structure $\mathcal{M}$ to abstract $\mathbb{K}$ in order to formalize some internal transformations that $\mathbb{K}$ does on semantics (Section 3). Thanks to the mathematical structure $\mathcal{M}$, we formalize a computational translation $\mathcal{T}$ into the $\lambda\Pi$-calculus modulo theory (Section 4). Finally, we present our implementation of $\mathcal{T}$ (Section 5).

In the following, the keywords of a language or what is native in a language will be distinguished by color. The language of Dedukti will be distinguished by a blue color, the language of 𝕂 by an orange color and the language of Kore by a red color. These facilitate reading but are not necessary for understanding.

2 What is the 𝕂 framework?

This section introduces the 𝕂 framework by explaining how to write a semantics using a small example, and then by presenting the diversity of 𝕂 features. This section ends with the 𝕂 grammar that we consider in the rest of this article.

2.1 A first 𝕂 semantics

In this subsection, we show an example based on booleans and two binary symbols, which are lazily evaluated, to illustrate how to write a semantics in 𝕂. As usual, the first step to formalize a semantics is to define the syntax and then the semantics associated to the syntax.

2.1.1 Define the syntax of a language

Defining the syntax of a language in 𝕂 is similar to writing a *BNF grammar*. This is done in the module LAZY-SYNTAX (Figure 2 - lines 1 to 9) with the definition of booleans, which are typed by sort **MyBool**, a lazy disjunction, noted `||`, and a lazy conjunction, noted `&&`. A terminal symbol will be written between quotes, as for example `"||"`, and anything else in **bold** will therefore be a non-terminal symbol. In order to make the syntax parseable, it is possible to use *attributes*, i.e. keywords between square brackets, allowing to specify the associativity (`left`, `right`, `non-assoc`) or to add parentheses to the language (`bracket`). We explain the other possible attributes as we go along. Moreover, 𝕂 supports the Kleene operators "*" and "+", written `List` and `NeList` respectively.

```
1   module LAZY-SYNTAX
2         syntax MyBool ::= "true"                  [ constructor ]
3                         | "false"                 [ constructor ]
4         syntax KResult ::= MyBool
5         syntax BExp ::= MyBool
6           | BExp "||" BExp                        [ left, function ]
7           | BExp "&&" BExp                        [ left, constructor, strict(1) ]
8           | "(" BExp ")"                          [ bracket ]
9   endmodule
10  module LAZY
11        imports LAZY-SYNTAX
12        configuration <k> $PGM : BExp </k>

13        rule false || B => B
14        rule true  || _ => true

15        rule true  && B => B
16        rule false && _ => false
17  endmodule
```

Figure 2 Syntax and semantics of booleans, a lazy disjunction and a lazy conjunction.

The subtyping relation between the sorts **MyBool** and **BExp** (Figure 2 - line 5) means that the boolean values `true` and `false` are also boolean expressions. In addition, any symbol has either the attribute `constructor`, when the symbol is an element of the syntax, or the attribute `function`, when the symbol is a *helper function* used to define the semantics, e.g. functions to manipulate the environment.

2.1.2 Define the semantics associated to the syntax

The main ingredients for defining the semantics of each element of the syntax are *configurations* and *rewriting rules*. Some attributes are also useful to define evaluation strategies. The semantics of a lazy disjunction and a lazy conjunction is defined in the module LAZY (Figure 2 - lines 10 to 17) which imports the syntax module (line 11 thanks to `imports`).

2.1.2.1 Configurations

A *configuration* models the state of the program and is composed of *cells*. For example, the configuration $\langle\langle\ x = 10;\ \rangle_k\ \langle\ x \mapsto 0\ \rangle_{env}\rangle$ is composed of two cells, one labelled by k containing the program to be executed and the other labelled by *env* containing the current values of the variables. In the example in Figure 2, the configuration contains only the cell k (line 12). The configuration variable `$PGM` will contain the parsed program given by the user.

2.1.2.2 Rewriting rules

A $\mathbb{K}$ rewriting rule is a 1st order rule which can be either conditional, noted `rule` *LHS* `=>` *RHS* `requires` *Cond*, or unconditional, noted `rule` *LHS* `=>` *RHS*. Moreover, a $\mathbb{K}$ rewriting rule can be non-linear, i.e. variables in the left-hand side can appear several times. The variables in the left-hand side (*LHS*) can be omitted using a wildcard (`_`) when they are not used in the right-hand side (*RHS*), as in the rule on line 14 or 16 (Figure 2). Finally, $\mathbb{K}$ supports partial rewriting modulo ACUI, i.e. associativity (`assoc`), commutativity (`comm`), identity (`unit`) and idempotence (`idem`).

Any $\mathbb{K}$ rewriting rule can be applied to a whole configuration, if the rewriting rule defines the semantics associated to the syntax, or does not mention the configuration, if the rewriting rule defines a helper function. This distinction is illustrated more precisely in the next paragraph.

2.1.2.3 Evaluation strategies

To define an evaluation strategy, i.e. specifying the order in which the sub-expressions are evaluated, it is possible to use *contexts* (`context`) as is conventionally done, but also *context aliases* (`context alias`) which allow contexts to be generated automatically rather than systematically writing similar contexts.

There are also two attributes for defining an evaluation strategy: `strict` defines non-deterministic strategies and `seqstrict` defines deterministic strategies from left to right by default. It is also possible to restrict the list of sub-expressions that must be evaluated by giving a list of numbers as done in Figure 2. Indeed, the attribute `strict(1)` forces the evaluation of the first argument of the symbol `&&`, and then it is possible to apply one of the rules on line 15 or 16. To use these attributes, the user needs to define the sort `KResult` which allows to distinguish final values from expressions thanks to subtyping. For instance, as **MyBool** is a sub-sort of `KResult` (Figure 2 - line 4), a final value is either `false` or `true`.

Whichever way an evaluation strategy is defined, it is translated using $\mathbb{K}$ *computations* and *freezers*. A $\mathbb{K}$ computation is a list of computations to be performed sequentially and built with the constructors `.` and $\curvearrowright$, whereas a freezer is a symbol that encapsulates the part of the computation that should not yet be modified, i.e. the tail of the $\mathbb{K}$ computation, while waiting for the head of the $\mathbb{K}$ computation to be evaluated. This mechanism is inspired by evaluation contexts [26] and continuations $v \curvearrowright C$.
The rewriting rules generated by `strict(1)` (Rules n°1 and n°2, with the attributes `heat` and `cool`) as well as an example of an execution are detailed in Figure 3. Freezers are noted

$(\text{❄}^{nb}_{sym}$ arg) where *sym* is a symbol, *nb* the number of the argument whose value we expect, and *arg* the list of other arguments. As the symbol `&&` has the attribute `constructor`, the rules on lines 15 and 16 (Figure 2) are respectively translated by 𝕂 into the rules n°3 and n°4 (Figure 3). In contrast, as the symbol `||` has the attribute `function`, so 𝕂 does not transform the rules on lines 13 and 14 (Figure 2). The translation of the attribute `strict(1)` into rewriting rules is similar in the case of attributes `strict` and `seqstrict`.

1. $\texttt{rule}\ \langle\ E_1\ \texttt{\&\&}\ E_2 \curvearrowright S\ \rangle_k \quad \texttt{=>}\ \langle\ E_1 \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ E_2) \curvearrowright S\ \rangle_k\ \texttt{requires}\ \neg\ (\texttt{isKResult}\ E_1)\ \texttt{[heat]}$
2. $\texttt{rule}\ \langle\ E_1 \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ E_2) \curvearrowright S\ \rangle_k\ \texttt{=>}\ \langle\ E_1\ \texttt{\&\&}\ E_2 \curvearrowright S\ \rangle_k \quad \texttt{requires}\quad \texttt{isKResult}\ E_1\ \texttt{[cool]}$
3. $\texttt{rule}\ \langle\ \texttt{true}\ \ \texttt{\&\&}\ B \curvearrowright S\ \rangle_k\ \ \texttt{=>}\ \ \langle\ B \curvearrowright S\ \rangle_k$
4. $\texttt{rule}\ \langle\ \texttt{false}\ \texttt{\&\&}\ \_ \curvearrowright S\ \rangle_k\ \ \texttt{=>}\ \ \langle\ \texttt{false} \curvearrowright S\ \rangle_k$

$$\begin{array}{rcr}
\langle\ (\texttt{true \&\& false})\ \texttt{\&\&}\ (\texttt{true \&\& true}) \curvearrowright\ .\ \rangle_k & & \langle\ e_1\ \texttt{\&\&}\ e_2 \curvearrowright s\ \rangle_k \\
\hookrightarrow_1 \langle\ (\texttt{true \&\& false}) \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ (\texttt{true \&\& true})) \curvearrowright\ .\ \rangle_k & & \hookrightarrow_1 \langle\ e_1 \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ e_2) \curvearrowright s\ \rangle_k \\
\hookrightarrow_3 \langle\ \texttt{false} \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ (\texttt{true \&\& true})) \curvearrowright\ .\ \rangle_k & \text{abstracted by} & \hookrightarrow_3 \langle\ v_1 \curvearrowright (\text{❄}^{1}_{\texttt{\&\&}}\ e_2) \curvearrowright s\ \rangle_k \\
\hookrightarrow_2 \langle\ \texttt{false}\ \texttt{\&\&}\ (\texttt{true \&\& true}) \curvearrowright\ .\ \rangle_k & & \hookrightarrow_2 \langle\ v_1\ \texttt{\&\&}\ e_2 \curvearrowright s\ \rangle_k \\
\hookrightarrow_4 \langle\ \texttt{false} \curvearrowright\ .\ \rangle_k & & \hookrightarrow_4 \langle\ v_1 \curvearrowright s\ \rangle_k
\end{array}$$

Figure 3 Translation of the attributes `strict(1)` and an example execution.

In this article, a rewriting rule that has a `constructor` symbol as its head is called *semantical*, and a rewriting rule that has a `function` symbol as its head is called *evaluation*. The attributes `assoc`, `comm`, `unit` and `idem` generate equations, named *equational rules*. A rewriting rule with the attribute `heat` or `cool` is called an *evaluation strategy rule*.

2.2 Additional features

The previous subsection illustrated the main 𝕂 features. However, there are many other features, coming from attributes or the 𝕂 standard library, in order to bring more precision to a semantics.

2.2.1 Definable features thanks to the attributes

𝕂 has about 70 attributes. Papers about 𝕂, e.g. [19], mention very few of them and the documentation [2] is not exhaustive and complete. However, many features require the use of attributes. This section presents the list of attributes in Figure 4 that we hope to be as exhaustive as possible.

About importation.	$\mathcal{A}_{library}$	$\triangleq$	{ `hook` }
	$\mathcal{A}_{visibility}$	$\triangleq$	{ `public`, `private` }
	$\mathcal{A}_{backend}$	$\triangleq$	{ `symbolic`, `concrete`, `kast`, `kore` }
About parsing.	$\mathcal{A}_{parsing}$	$\triangleq$	{ `left`, `right`, `non-assoc`, `prefer`, `avoid`, `applyPriority` }
	$\mathcal{A}_{sort}$	$\triangleq$	{ `token`, `locations`, `hook` }
	$\mathcal{A}_{token}$	$\triangleq$	{ `prefer`, `prec(`*nb*`)`, `hook` }
About printing.	$\mathcal{A}_{printing}$	$\triangleq$	{ `color`, `colors`, `symbol`, `klabel`, `bracketLabel`, `format`, `latex`, `unused` }
About symbol.	$\mathcal{A}_{family}$	$\triangleq$	{ `constructor`, `function`, `token`, `bracket`, `macro` }
	$\mathcal{A}_{property}$	$\triangleq$	{ `injective`, `total`, `freshGenerator`, `binder` }
	$\mathcal{A}_{strategy}$	$\triangleq$	{ `strict`, `seqstrict`, `result`, `hybrid` }
About cell.	$\mathcal{A}_{structure}$	$\triangleq$	{ `multiplicity=` {`"+"` | `"*"` }, `type="` ⟨**sort**⟩`"` }
	$\mathcal{A}_{console}$	$\triangleq$	{ `exit="` ⟨**sort**⟩`"`, `stream=` {`"stdin"` | `"stdout"` | `"stderr"` } }
About rewriting.	$\mathcal{A}_{modulo}$	$\triangleq$	{ `assoc`, `comm`, `unit`, `idem` }
	$\mathcal{A}_{rule}$	$\triangleq$	{ `heat`, `cool`, `priority(`*nb*`)`, `owise`, `anywhere`, `unboundVariables` }
About proof.	$\mathcal{A}_{\text{KProver}}$	$\triangleq$	{ `symbolic`, `concrete`, `all-path`, `one-path`, `simplification`, `trusted`, `smtlib`, `smt-lemma`, `smt-hook`, `memo` }

Figure 4 List of 𝕂 attributes as exhaustive as possible.

About importation. What comes from the 𝕂 standard library, briefly presented in Section 2.2.2, has the attribute `hook`. Furthermore, we can specify the visibility of a module or an import ($\mathcal{A}_{visibility}$) or that a module is only useful for some backends ($\mathcal{A}_{backend}$).

About parsing. The user can precise constraints about parsing such as the associativity of symbols (`left`, `right`, `non-assoc`) but also to reject cases of parsing ambiguity (`prefer`, `avoid`, `applyPriority`). Moreover, it is possible to type a part of the AST by declaring particular identifiers (`token`) that can be used later in the semantics. The precedence of a token is given by the attribute `prec(`*nb*`)`. Sorts with the attribute `token` are only composed of symbols with the attribute `function` or `token`, and only these sorts can be composed of `token` symbols. Finally, 𝕂 is able to insert file, line and column meta-data into the parse tree on a subtree of type s when parsing, when the sort s has the attribute `locations`.

About printing. There are also some attributes to change colors of printing in the console (`color`, `colors`), the names (`symbol`, `klabel`, `bracketLabel`) and the printing (`format`) of the symbols and to define a latex name (`latex`). Moreover, 𝕂 will warn the user if a symbol is declared but not used in any of the rules. The user can disable this warning for a particular symbol or cell by adding the attribute `unused`.

About symbol. Compared to an evaluation strategy defined with the attributes `strict` and `seqstrict` (Section 2.1.2.3), it is possible to develop more complex ones thanks to the attributes `result` and `hybrid`. For example, these attributes can allow lists of values to be considered as values.

A symbol can be (1) a *constructor*, (2) a *function*, (3) a *token*, (4) a *bracket* or (5) a *macro* ($\mathcal{A}_{family}$). Functions can be defined as injective (`injective`) or total (`total`, formerly called `functional`). Moreover, it is possible to request 𝕂 to generate fresh values and use them with fresh variables `!`Var (`freshGenerator`) and also to define binder (`binder`).

About cell. The user can choose if a cell is optional (`multiplicity="?"`) or can appear several times (`multiplicity="*"`). These attributes allow the user to design a set of cells, which type can be defined by sorts `List`, `Set` or `Map` thanks to the attribute `type`. Moreover, each cell can have a console exit value (`exit`) or can print on the standard stream (`stream`).

About rewriting. Theoretically rewriting rules can be applied in any order, but 𝕂 allows the user to associate a priority to each rule (`priority(`*nb*`)`) or to indicate that a rule applies only if no other can apply (`owise`). Moreover, the attribute `anywhere` can be used to prevent 𝕂 from automatically computing the configuration in a rewriting rule. Finally, it is also possible to allow variables to be unbound in the left-hand side of a rewriting rule thanks to the attribute `unboundVariables` or with unbound variables `?`Var.

The semantics of most of these attributes are specified in the rest of this article.

2.2.2 Definable features thanks to the 𝕂 standard library

The 𝕂 standard library is composed of eight files. The file `prelude.md` is imported into any 𝕂 definition and contains only two lines that import the following two files: `domains.md` which defines the types of several usual data structures, for instance, the sorts `Bytes`, `Array`, `Map`, `Set`, `List`, `Bool`, `Int`, `String` or `Float`, and `kast.md` which corresponds to the syntax of 𝕂. Moreover, the file `rat.md` is an implementation of the rational integers, the file substitution.md is an implementation allowing substitution (required by the attribute binder), and the file unification.md is an implementation allowing unification. Finally, the two last files are ffi.md which allows C functions to be called and json.md which allows JSON files to be read. The three following symbols are also defined: . : K [constructor], $\curvearrowright$: KItem × K → K [constructor] and inj : $\forall$ (*From*, *To* : K), *From* → *To*.

2.3 A $\mathbb{K}$ grammar

To conclude this section, we present the overview of the translation from $\mathbb{K}$ semantics which is available in Figure 5 and formalized in the next section. The $\mathbb{K}$ grammar that we consider in this article is available in Figure 6. A $\mathbb{K}$ file can contain several modules (module/endmodule). It is possible to import files into another file (requires or require) or to import one or more modules into another module (imports or import). Additionally, an attribute is associated to a module ($\mathcal{A}_{module}$), a sort ($\mathcal{A}_{sort}$), a symbol ($\mathcal{A}_{symbol}$), a cell ($\mathcal{A}_{cell}$), a rewriting rule ($\mathcal{A}_{rule}$), a context ($\mathcal{A}_{context}$) or a context alias ($\mathcal{A}_{context-alias}$). Moreover, a configuration variable begins with $ such as $PGM, a fresh variable begins with ! and an unbound variable in a rewriting rule begins with ?. Finally, there are three cast operators noted :, :: and :>.

The $\mathbb{K}$ grammar in Figure 6 is almost complete. In order not to make it too heavy, we have omitted a part of the syntax allowing to declare precedence and associativity of symbols since this is only useful for the generation of the $\mathbb{K}$ parser extended with the user-defined language $\mathcal{L}$ (Figure 5).

In addition, we consider that the declarations with at least one of the attributes related to symbolic execution ($\mathcal{A}_{\text{KProver}}$) have been deleted as well as the syntactic sugar has been simplified. For example,
syntax **BExp** ::= **MyBool** | **BExp** "&&" **BExp** is syntactic sugar for
syntax **BExp** ::= **MyBool**
syntax **BExp** ::= **BExp** "&&" **BExp**. We do not take into account either the syntax ... or the fact that the rewriting arrow => can be nested following Li and Gunter [15] who explain that this syntax is ambiguous syntactic sugar. We assume that the syntactic sugar is simplified by the black box "Desugar" (Figure 5).

In this article, we only consider a $\mathbb{K}$ semantics if $\mathbb{K}$ accepts it as well as the $\mathbb{K}$ grammar (Figure 6) – after deleting a part of the syntax and the syntactic sugar.

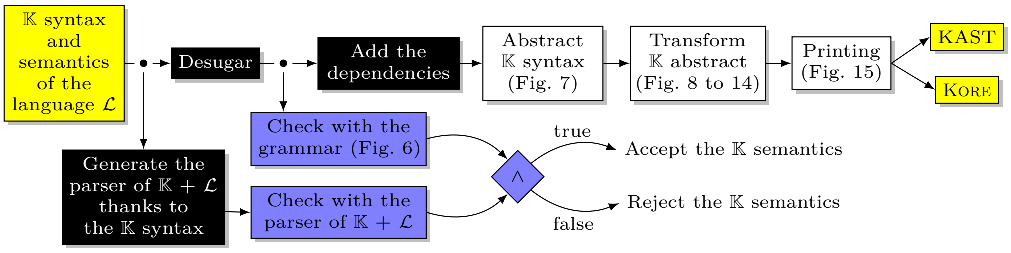

$\mathbb{K}$ syntax and semantics of the language $\mathcal{L}$
Desugar
Add the dependencies
Abstract $\mathbb{K}$ syntax (Fig. 7)
Transform $\mathbb{K}$ abstract (Fig. 8 to 14)
Printing (Fig. 15)
KAST
KORE
Check with the grammar (Fig. 6)
Generate the parser of $\mathbb{K}$ + $\mathcal{L}$ thanks to the $\mathbb{K}$ syntax
Check with the parser of $\mathbb{K}$ + $\mathcal{L}$
$\wedge$
true
false
Accept the $\mathbb{K}$ semantics
Reject the $\mathbb{K}$ semantics

Figure 5 Overview of the translation from $\mathbb{K}$ semantics.

We just explained the two first black boxes in Figure 5. Other boxes are explained (black one) or formalized (white one) in the next section.

⟨carac⟩ ::= [a-zA-Z | 0-9 | - | _]
⟨int⟩ ::= [1-9][0-9]*
⟨string⟩ ::= " ⟨carac⟩* "
⟨name-module⟩ ::= ⟨carac⟩+
⟨symbol⟩ ::= ⟨carac⟩+
⟨str-of-reg-expr⟩ ::= a regular expression between quotes

⟨require⟩ ::= (require | requires) " ⟨carac⟩+ [.k | .md] "
⟨import⟩ ::= (import | imports) [public | private]? ⟨name-module⟩

⟨sort⟩ ::= [A-Z #] ⟨carac⟩*
⟨sort-syntax⟩ ::= syntax ⟨sort⟩ ([token | locations | $\boxed{\mathcal{A}_{sort}}^{+}_{,}$])?
| syntax ⟨sort⟩ ::= ⟨sort⟩ ([token])?

⟨terminal⟩ ::= ⟨string⟩
⟨non-terminal⟩ ::= ⟨sort⟩
⟨syntax-item⟩ ::= ⟨terminal⟩ | ⟨non-terminal⟩
⟨separator⟩ ::= ⟨string⟩
⟨syntax⟩ ::= syntax [{ $\langle\text{sort}\rangle^{+}_{,}$ }]? ⟨sort⟩ ::= ⟨symbol⟩($\langle\text{sort}\rangle^{*}_{,}$) [$\boxed{\mathcal{A}_{symbol}}^{*}_{,}$]
| syntax [{ $\langle\text{sort}\rangle^{+}_{,}$ }]? ⟨sort⟩ ::= ⟨syntax-item⟩+ [$\boxed{\mathcal{A}_{symbol}}^{*}_{,}$]
| syntax ⟨sort⟩ ::= r ⟨str-of-reg-expr⟩ [{ token } ∪ $\boxed{\mathcal{A}_{token}}^{*}_{,}$]
| syntax ⟨sort⟩ ::= List { ⟨sort⟩ , ⟨separator⟩ }
| syntax ⟨sort⟩ ::= NeList { ⟨sort⟩ , ⟨separator⟩ }

⟨config-variable⟩ ::= $[A-Z]+
⟨initial-value⟩ ::= ⟨symbol⟩ | ⟨config-variable⟩
⟨cell⟩ ::= < ⟨name⟩ $\boxed{\mathcal{A}_{cell}}^{*}_{,}$> ⟨initial-value⟩ [: ⟨sort⟩]? < / ⟨name⟩>
| < ⟨name⟩ $\boxed{\mathcal{A}_{cell}}^{*}_{,}$> ⟨cell⟩+ < / ⟨name⟩>
⟨configuration⟩ ::= configuration ⟨cell⟩+

⟨variable⟩ ::= [? | !]? [A-Z] ⟨carac⟩* | _
⟨pattern⟩ ::= (⟨variable⟩ | ⟨symbol⟩) [: ⟨sort⟩ | :: ⟨sort⟩]?
| { ⟨pattern⟩+ } :> ⟨sort⟩
⟨rule⟩ ::= rule ⟨pattern⟩+ => ⟨pattern⟩+ [requires ⟨pattern⟩+]? [$\boxed{\mathcal{A}_{rule}}^{*}_{,}$]

⟨context⟩ ::= context ⟨pattern⟩+ [requires ⟨pattern⟩+]? ([$\boxed{\mathcal{A}_{context}}^{+}_{,}$])?
⟨context-alias⟩ ::= context alias [⟨carac⟩+]: ⟨pattern⟩+ ([$\boxed{\mathcal{A}_{context-alias}}^{+}_{,}$])?

⟨sentence⟩ ::= ⟨sort-syntax⟩ | ⟨syntax⟩
| ⟨configuration⟩ | ⟨rule⟩
| ⟨context⟩ | ⟨context-alias⟩

⟨module⟩ ::= module ⟨name-module⟩ ([$\boxed{\mathcal{A}_{module}}^{+}_{,}$])?
⟨import⟩*
⟨sentence⟩*
endmodule

⟨file⟩ ::= ⟨require⟩* ⟨module⟩*

where $\mathcal{A}_{module} \triangleq \mathcal{A}_{visibility} \cup \mathcal{A}_{backend} \cup$ { not-lr1 }
$\mathcal{A}_{symbol} \triangleq \mathcal{A}_{parsing} \cup \mathcal{A}_{family} \cup \mathcal{A}_{property} \cup \mathcal{A}_{strategy} \cup \mathcal{A}_{modulo} \cup \mathcal{A}_{printing} \cup \mathcal{A}_{visibility}$
$\mathcal{A}_{cell} \triangleq \mathcal{A}_{structure} \cup \mathcal{A}_{console} \cup$ { unused="", color= ⟨string⟩ }
$\mathcal{A}_{context} \triangleq$ { result }
$\mathcal{A}_{context-alias} \triangleq$ { result, context }

Figure 6 The considered 𝕂 grammar, where $\boxed{X}$ is any element of X.

3 Abstracting the $\mathbb{K}$ framework

This section presents a mathematical structure $\mathcal{M}$ which abstracts the syntax of $\mathbb{K}$. After the presentation of the structure $\mathcal{M}$ as well as the translation of the $\mathbb{K}$ syntax into the structure $\mathcal{M}$, we present various transformations of the structure $\mathcal{M}$ that correspond to the static semantics of $\mathbb{K}$. To present this work, we start with the output of the black box "Add the dependencies" (Figure 5) that recursively replaces each `require` command with the contents of these files, and then recursively replaces each `import` command with the contents of the module that must be imported. We consider that the output of the black box is a single module whose name corresponds to the name of the main initial semantics file. At this stage, the following attributes have finished influencing the transformation: `hook`, $\mathcal{A}_{visibility}$, $\mathcal{A}_{backend}$, `not-lr1`, $\mathcal{A}_{parsing}$, `token`, $\mathcal{A}_{token}$, `locations`, $\mathcal{A}_{printing}$ and $\mathcal{A}_{\text{KProver}}$.

3.1 An abstract view of $\mathbb{K}$

To abstract a $\mathbb{K}$ file, we use the 7-uplet $\mathcal{M} \triangleq (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}, \mathcal{C}ontext, \mathcal{C}ontext_{alias})$, where $\mathcal{S}ort$ is the set of sorts, $\mathcal{R}el$ is the set of subtyping relations, $\mathcal{S}ym$ is the set of symbols, $\mathcal{C}onfig$ is the set of configurations, $\mathcal{R}$ is the set of rewriting rules, $\mathcal{C}ontext$ is the set of contexts and $\mathcal{C}ontext_{alias}$ is the set of context aliases. Figure 7 shows the translation $||\ .\ ||$ of $\mathbb{K}$ syntax to the abstract structure $\mathcal{M}$. Syntactic declarations can create sorts, subtyping relations between two sorts, or symbols. As $\mathbb{K}$ allows mixfix notations, we translate them into prefix notations such as || `syntax` **Exp** `::= "if" Bool "then"` **Exp** `"else"` **Exp** || = || `syntax` **Exp** `::= if-then-else(Bool,` **Exp**, **Exp**`)` ||. Moreover, a configuration declaration generates a list of trees l. Section 3.2 shows that l is transformed into a single tree. Finally, any unconditional rewriting rule can be seen as a conditional rule with the condition "true".

$$
\begin{array}{l}
||\ \texttt{syntax}\ s\qquad\qquad (\ \texttt{[}\ Attr\ \texttt{]}\ )?\ || = \mathcal{S}ort \leftarrow \{\ s\ \} \\
||\ \texttt{syntax}\ s_1\ \texttt{::=}\ s_2\ (\ \texttt{[}\ Attr\ \texttt{]}\ )?\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \}\ ;\ \mathcal{R}el \leftarrow \{\ s_2 < s_1\ \} \\
||\ \texttt{syntax}\ \texttt{\{}\ \alpha_1,\ ...,\ \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ sym\ \texttt{(}\ s_1,\ ...,\ s_x\ \texttt{)}\ \texttt{[}\ Attr\ \texttt{]}\ ||\ \text{with } n \geq 0 \text{ and } x \geq 0 = \\
\quad \mathcal{S}ort \leftarrow \{\ \alpha\ ;\ s_1\ ;\ ...\ ;\ s_x\ \} \setminus \{\ \alpha_1,\ ...,\ \alpha_n\ \}\ ; \\
\quad \mathcal{S}ym \leftarrow \{\ sym : \forall\alpha_1, ..., \forall\alpha_n,\ s_1 \times ... \times s_x \rightarrow \alpha\ [Attr]\ \} \\
||\ \texttt{syntax}\ \texttt{\{}\ \alpha_1,\ ...,\ \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ i_1\ ...\ i_x\ \texttt{[}\ Attr\ \texttt{]}\ ||\ \text{with } n \geq 0 \text{ and } x \geq 1 = \\
\quad ||\ \texttt{syntax}\ \texttt{\{}\ \alpha_1,\ ...,\ \alpha_n\ \texttt{\}}\ \alpha\ \texttt{::=}\ t_1\text{-}...\text{-}t_n\ \texttt{(}\ s_1,\ ...,\ s_x\ \texttt{)}\ \texttt{[}\ Attr\ \texttt{]}\ || \\
\qquad \text{where } t_i \in I_{terminal} \qquad\quad \triangleq \{\ i_k \mid k \in [\![1; x]\!] \text{ and } i_k \in \langle\textbf{terminal}\rangle\ \} \\
\qquad\qquad\quad s_i \in I_{non-terminal} \triangleq \{\ i_k \mid k \in [\![1; x]\!] \text{ and } i_k \in \langle\textbf{non-terminal}\rangle\ \} \\
||\ \texttt{syntax}\ s\ \ \texttt{::= r}\ \langle\textbf{str-of-reg-expr}\rangle\ \texttt{[}\ Attr\ \texttt{]}\ || = \mathcal{S}ort \leftarrow \{\ s\ \} \\
||\ \texttt{syntax}\ s_1\ \texttt{::= List}\quad\ \texttt{\{}\ s_2\texttt{,}\ sep\ \texttt{\}}\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \} \\
||\ \texttt{syntax}\ s_1\ \texttt{::= NeList}\ \texttt{\{}\ s_2\texttt{,}\ sep\ \texttt{\}}\ || = \mathcal{S}ort \leftarrow \{\ s_1\ ;\ s_2\ \} \\
\\
||\ \texttt{configuration}\ cell_1\ ...\ cell_n\ ||\ \text{with } n \geq 1 = \mathcal{C}onfig \leftarrow \{\ (||\ cell_1\ ||_{\texttt{rec}}\ ;\ ...\ ;\ ||\ cell_n\ ||_{\texttt{rec}})\ \} \\
||\ \texttt{<}\ C\ \texttt{>}\ v\ \texttt{:}\ s\ \texttt{</}\ C\ \texttt{>}\ ||_{\texttt{rec}} \qquad\qquad = [C]_s(v) \qquad\qquad \text{(If } s \text{ is not given, we can infer it from } v\text{.)} \\
||\ \texttt{<}\ C\ \texttt{>}\ cell_1\ ...\ cell_n\ \texttt{</}\ C\ \texttt{>}\ ||_{\texttt{rec}} = \texttt{<}C\texttt{>}(||\ cell_1\ ||_{\texttt{rec}},\ ...,\ ||\ cell_n\ ||_{\texttt{rec}}) \\
\\
||\ \texttt{rule}\ LHS\ \texttt{=>}\ RHS\ \texttt{[}\ Attr\ \texttt{]}\ || \qquad\qquad\qquad\qquad\quad = \mathcal{R} \leftarrow \{\ (LHS \overset{\texttt{true}}{\hookrightarrow} RHS\ [Attr])\ \} \\
||\ \texttt{rule}\ LHS\ \texttt{=>}\ RHS\ \texttt{requires}\ Cond\ \texttt{[}\ Attr\ \texttt{]}\ || = \mathcal{R} \leftarrow \{\ (LHS \overset{Cond}{\hookrightarrow} RHS\ [Attr])\ \} \\
\\
||\ \texttt{context}\ C\ (\texttt{[}\ Attr\ \texttt{]})?\ || \qquad\qquad\qquad\qquad = \mathcal{C}ontext \leftarrow \{\ (C,\ \texttt{true},\ Attr)\ \} \\
||\ \texttt{context}\ C\ \texttt{requires}\ Cond\ (\texttt{[}\ Attr\ \texttt{]})?\ || \qquad = \mathcal{C}ontext \leftarrow \{\ (C,\ Cond,\ Attr)\ \} \\
||\ \texttt{context alias}\ \texttt{[}\ label\ \texttt{]:}\ CA\ (\texttt{[}\ Attr\ \texttt{]})?\ || = \mathcal{C}ontext_{alias} \leftarrow \{\ (label,\ CA,\ Attr)\ \}
\end{array}
$$

Figure 7 From $\mathbb{K}$ to an abstract 7-uplet, where $X \leftarrow data$ is the set X extended with $data$.

The following subsection explains the internal transformations made by $\mathbb{K}$ from the obtained mathematical structure $\mathcal{M}$.

3.2 Compilation of a $\mathbb{K}$ semantics

This subsection formalizes the transformations on the previous mathematical structure $\mathcal{M}$ that correspond to the static semantics of $\mathbb{K}$. After these transformations, the 7-uplet $\mathcal{M}$ will become a quadruplet. As we abstract the content of the $\mathbb{K}$ standard library by the sets $\mathcal{S}ort_{lib}$, $\mathcal{R}el_{lib}$ $\mathcal{S}ym_{lib}$ and $\mathcal{R}_{lib}$, we initially assume that $\mathcal{S}ort = \mathcal{S}ort_{lib}$, $\mathcal{R}el = \mathcal{R}el_{lib}$, $\mathcal{S}ym = \mathcal{S}ym_{lib}$ and $\mathcal{R} = \mathcal{R}_{lib}$ whereas $\mathcal{C}onfig$, $\mathcal{C}ontext$ and $\mathcal{C}ontext_{alias}$ are empty. The goal of the first transformation (Figure 8) is to check the validity of a $\mathbb{K}$ semantics.

Each symbol should be a *constructor*, a *function*, a *token*, a *bracket* or a *macro*. If this is not the case, it means that the symbol is implicitly a constructor. We therefore explicitly add the attribute `constructor`, which defines the new set $\mathcal{S}ym'$ (Figure 8). Regarding the attribute `macro`, the documentation is not precise enough. According to Li and Gunter [14], macros are subject to many errors when writing semantics, but in practice macros are always considered as syntactic sugar. The associated rewriting rules are used only once at the beginning of an evaluation of the input programs, to rewrite the syntactic sugar into another term. Macros can therefore be replaced by functions as they are not more expressive.

Moreover, a semantics must have only one configuration, and some attributes have precise restrictions such as the attributes `strict` and `seqstrict` are incompatible with `function`, while a `bracket` symbol of a given sort has only one argument of that sort.

$$
\begin{array}{l}
||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{C}onfig,\ \mathcal{R},\ \mathcal{C}ontext,\ \mathcal{C}ontext_{alias})\ ||_{\texttt{check-input}} = \\
\quad (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym',\ \mathcal{C}onfig',\ \mathcal{R},\ \mathcal{C}ontext,\ \mathcal{C}ontext_{alias}) \\
\qquad \text{where} \\
\qquad\qquad \begin{array}{lcl}
\mathcal{C} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{constructor} \in Attr(s)\ \} \\
\mathcal{F} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{function} \in Attr(s)\ \} \\
\mathcal{T} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{token} \in Attr(s)\ \} \\
\mathcal{B} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{bracket} \in Attr(s)\ \} \\
\mathcal{M}a & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{macro} \in Attr(s)\ \} \\
\mathcal{W} & \triangleq & \mathcal{S}ym \setminus (\mathcal{C}\ \cup\ \mathcal{F}\ \cup\ \mathcal{T}\ \cup\ \mathcal{B}\ \cup\ \mathcal{M}a) \\
\mathcal{W}' & \triangleq & \{\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr \cup \{\texttt{constructor}\}] \mid\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{W}\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{C}\ \cup\ \mathcal{F}\ \cup\ \mathcal{T}\ \cup\ \mathcal{B}\ \cup\ \mathcal{M}a\ \cup\ \mathcal{W}' \\
\mathcal{C}onfig' & \triangleq & [k]_{\texttt{K}}(\texttt{\$PGM}) \text{ if } \mathcal{C}onfig = \emptyset;\ \mathcal{C}onfig \text{ if } \mathrm{card}(\mathcal{C}onfig) = 1
\end{array} \\
\quad \textbf{Constraints:} \\
\qquad\qquad \text{The set } \{\ \mathcal{C},\ \mathcal{F},\ \mathcal{T},\ \mathcal{B},\ \mathcal{M}a,\ \mathcal{W}\ \} \text{ must be a partition of } \mathcal{S}ym. \\
\qquad\qquad \mathcal{C}onfig \text{ is the empty set or a singleton, where each cell have a unique name.} \\
\qquad\qquad \text{For all } n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{F},\ \{\ \texttt{strict},\ \texttt{seqstrict}\ \} \cap Attr = \emptyset. \\
\qquad\qquad \text{For all } n : \forall \overrightarrow{v},\ \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{B},\ \overrightarrow{v} = \overrightarrow{0} \text{ and } \overrightarrow{t} = \alpha.
\end{array}
$$

Figure 8 Formalization of the function $||\ .\ ||_{\texttt{check-input}}$.

Generate evaluation strategy rules thanks to contexts and context aliases. We have seen that to define evaluation strategies in $\mathbb{K}$, we can define context aliases, contexts or use the attributes `strict` and `seqstrict`. We assume the existence of a function $||\ .\ ||_{\texttt{delete-context-alias}}$ that transforms a context alias into contexts as well as a function $||\ .\ ||_{\texttt{delete-context}}$ that transforms a context into rewriting rules. The lack of information in the documentation prevents us from formalizing these two functions.

Generate evaluation strategy rules thanks to attributes. We formalize the rule generation from the attributes `strict` and `seqstrict` (when the attributes `result` and `hybrid` are not also used) in Figure 9, where `nbArg`(sym) is the number of argument(s) of the symbol sym and n is implicitly used to denote the arity of a symbol. For example, rules 1. and 2. (Figure 3) illustrate the translation formalized in Figure 9.

$$|| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}) ||_{\texttt{generate-strategy}} = (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym', \mathcal{C}onfig, \mathcal{R}')$$

where

$$\begin{array}{lcl}
\mathcal{S}_{\texttt{strict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{strict} \in Attr(s)\ \} \\
\mathcal{S}_{\texttt{seqstrict}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{seqstrict} \in Attr(s)\ \} \\
\mathcal{F}reezer & \triangleq & \{\ \text{❄}^{n_j}_{sym} :\ \underbrace{\texttt{K} \times \ldots \times \texttt{K}}_{\texttt{nbArg}(sym)-1}\ \to \texttt{KItem}\ [\texttt{constructor}\ ] \\
 & & \qquad \mid\ \text{and } \texttt{strict}\ (n_1,\ldots,n_k) \in Attr(sym) \\
 & & \qquad\ \ \ \text{or } \texttt{seqstrict}\ (n_1,\ldots,n_k) \in Attr(sym)\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{S}ym\ \cup\ \mathcal{F}reezer \\
\mathcal{R}' & \triangleq & \mathcal{R}\ \cup\ \mathcal{R}_{\texttt{strict}}\ \cup\ \mathcal{R}_{\texttt{seqstrict}}
\end{array}$$

Constraint:

Every argument of the attribute strict or seqstrict should be in $[1; arity(sym)]$.

$\mathcal{R}_{\texttt{strict}}$ is composed of the following rewriting rules:

For all $sym \in \mathcal{S}_{\texttt{strict}}$ such that $\texttt{strict}\ (n_1,\ldots,n_j) \in Attr(sym)$, for all $k \in \{\ n_1,\ \ldots,\ n_j\ \}$:

$$sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\text{❄}^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ],\ \text{where } c \triangleq \neg\ (\texttt{isKResult}\ E_k)$$

$$E_k \curvearrowright (\text{❄}^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ],\ \text{where } c \triangleq \texttt{isKResult}\ E_k$$

$\mathcal{R}_{\texttt{seqstrict}}$ is composed of the following rewriting rules:

For all $sym \in \mathcal{S}_{\texttt{seqstrict}}$ such that $\texttt{seqstrict}\ (n_1,\ldots,n_j) \in Attr(sym)$, for all $k \in \{\ n_1,\ \ldots,\ n_j\ \}$:

$$sym\ E_1\ \ldots\ E_n \overset{c}{\hookrightarrow} E_k \curvearrowright (\text{❄}^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n)\ [\texttt{heat}\ ]$$

$$\text{where } c \triangleq \texttt{isKResult}\ E_1 \wedge \ldots \wedge \texttt{isKResult}\ E_{k-1} \wedge \neg\ (\texttt{isKResult}\ E_k)$$

$$E_k \curvearrowright (\text{❄}^{k}_{sym}\ E_1\ \ldots\ E_{k-1}\ E_{k+1}\ \ldots\ E_n) \overset{c}{\hookrightarrow} sym\ E_1\ \ldots\ E_n\ [\texttt{cool}\ ],\ \text{where } c \triangleq \texttt{isKResult}\ E_k$$

Figure 9 Formalization of the function $||\ .\ ||_{\texttt{generate-strategy}}$.

Encapsulate the configuration. According to the grammar in Figure 6, a configuration is a list of finite branching trees. However $\mathbb{K}$ encapsulates any configuration $[cell_1;\ldots;\ cell_n]$ as follows: $<\text{GT}>(<\text{T}>(cell_1,\ldots,\ cell_n),\ [\text{GC}]_{\texttt{Int}}(0))$, where GT ≜ `GeneratedTop` and GC ≜ `GeneratedCounter`. For instance, the initial configuration from Figure 2 becomes $\langle\ \langle\ \langle\ \texttt{\$PGM : }\textbf{BExp}\ \rangle_k\ \rangle_T\ \langle\ 0\ \rangle_{GC}\ \rangle_{GT}$.
Thus, after this transformation, any configuration becomes a single finite branching tree.

Generate implicit cells. Now that we have generated the full configuration, we formalize the completion of the rewriting rules in Figure 10. For instance, the result of mgconf $\mathcal{C}onfig$ is $\langle\ \langle\ \langle\ pgm\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$, where $\mathcal{C}onfig$ is the initial configuration of Figure 2, pgm and c are fresh variables. Moreover, rule 4. from Figure 3 becomes:
$\texttt{rule}\ \langle\ \langle\ \langle\ \texttt{false \&\& \_} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}\ \texttt{=>}\ \langle\ \langle\ \langle\ \texttt{false} \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$.

$$|| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}) ||_{\texttt{add-cell}} = (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{C}onfig, \mathcal{R}')$$

where

$$\begin{array}{ll}
 & \mathcal{R}' \ \triangleq\ \{\ ||\ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ and } x \triangleq \text{mgconf } \mathcal{C}onfig\ \} \\
\text{with} & \text{mgconf } (<C>(Cell_1,\ \ldots,\ Cell_n)) \triangleq\ <C>(\text{mgconf } Cell_1,\ \ldots,\ \text{mgconf } Cell_n) \\
 & \text{mgconf } ([C]_s(v)) \triangleq [C]_s(f),\ \text{where } f \text{ is a fresh variable} \\
\text{with} & ||\ l \overset{c}{\hookrightarrow} r\ [Attr]\ ||_x\ \ \triangleq l \overset{c}{\hookrightarrow} r\ [Attr],\ \text{if the head symbol of } l \text{ is a function symbol} \\
 & \qquad\qquad\qquad\quad\ \ l \overset{c}{\hookrightarrow} r\ [Attr],\ \text{if } \texttt{anywhere} \in Attr \\
 & \qquad\qquad\qquad\quad\ \ \overline{l}^{\,x} \overset{c}{\hookrightarrow} \overline{r}^{\,x}\ [Attr],\ \text{otherwise} \\
\text{with} & \overline{p}^{\,x} \triangleq x\ [\ \overrightarrow{[C]_s(y)} \setminus \overrightarrow{[C]_s(v)}\ ] \text{ where } \overrightarrow{[C]_s(v)} \text{ are the leaves appearing in } p
\end{array}$$

Figure 10 Formalization of the function $||\ .\ ||_{\texttt{add-cell}}$.

Split the configuration. Now we are ready to decompose the configuration into sorts and symbols. The generated rewriting rules are useful to complete the initial configuration with the value given by the user thanks to the configuration variable. The formalization is available in Figure 11, where GT ≜ `GeneratedTop` and GC ≜ `GeneratedCounter`.

$$
\begin{array}{lll}
\multicolumn{3}{l}{||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{C}onfig,\ \mathcal{R})\ ||_{\texttt{split-config}} = (\mathcal{S}ort',\ \mathcal{R}el,\ \mathcal{S}ym',\ \mathcal{R}')} \\
\text{where} & & \\
\mathcal{S}ort' & \triangleq & \mathcal{S}ort \cup \{\ \texttt{Sort}C \mid <C>(c_1,\ ...,\ c_n) \in \mathcal{C}onfig\ \} \\
\mathcal{S}ym_{cell} & \triangleq & \{\ C : \text{Type}\ (\ <C>(c_1,\ ...,\ c_n)\ )\ [\texttt{constructor}\ ] \mid <C>(c_1,\ ...,\ c_n) \in \mathcal{C}onfig\} \\
\mathcal{S}ym_{init} & \triangleq & \{\ \texttt{init}C : \texttt{Sort}C\ [\texttt{function},\ \texttt{initializer}\ ] \mid <C>(c_1,\ ...,\ c_n) \in \mathcal{C}onfig\} \\
\mathcal{S}ym_{get} & \triangleq & \{\ \texttt{get}\text{GC} : \texttt{Sort}\text{GT} \rightarrow \texttt{Sort}\text{GC}\ [\texttt{function}\ ]\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{S}ym\ \cup\ \mathcal{S}ym_{cell}\ \cup\ \mathcal{S}ym_{init}\ \cup\ \mathcal{S}ym_{get} \\
\mathcal{R}_{init} & \triangleq & \{\ \texttt{init}C \hookrightarrow C((\text{GetInit}\ c_1),\ ...,\ (\text{GetInit}\ c_n))\ [\texttt{initializer}\ ] \\
 & & \quad \mid <C>(c_1,\ ...,\ c_n) \in \mathcal{C}onfig\} \\
\mathcal{R}' & \triangleq & \mathcal{R}\ \cup\ \mathcal{R}_{init} \cup\ \{\ \texttt{get}\text{GC}(\text{GT}(\text{X,V})) \hookrightarrow \text{V}\ \} \\
\text{with} & \multicolumn{2}{l}{\text{Type}\ (\ <X>(c_1,\ ...,\ c_n)\ ) \triangleq \text{RetType}(c_1) \times ... \times \text{RetType}(c_n) \rightarrow \texttt{Sort}X} \\
\text{with} & \multicolumn{2}{l}{\text{RetType}\ (\ <X>(c_1,\ ...,\ c_n)\ ) \triangleq \texttt{Sort}X} \\
 & \multicolumn{2}{l}{\text{RetType}\ (\ [X]_s(v)\ ) \triangleq s} \\
\text{with} & \multicolumn{2}{l}{\text{GetInit}\ (\ <X>(c_1,\ ...,\ c_n)\ ) \triangleq \texttt{init}X} \\
 & \multicolumn{2}{l}{\text{GetInit}\ (\ [X]_s(v)\ ) \triangleq v}
\end{array}
$$

Figure 11 Formalization of the function $||\ .\ ||_{\texttt{split-config}}$.

For example, the symbol `initK : SortK` and the rule `initK` $\hookrightarrow$ `K $PGM` are generated as well as the symbols GT : `SortT` $\times$ `Sort`GC $\rightarrow$ `Sort`GT and `T` : `SortK` $\rightarrow$ `SortT`.

Add implicit attributes. Implicitly, every symbol with the attribute `constructor` also has the attributes `total`, formerly called `functional`, and `injective`, as formalized in Figure 12.

$$
\begin{array}{lll}
\multicolumn{3}{l}{||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\ ||_{\texttt{add-attributes}} = (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym',\ \mathcal{R})} \\
\text{where} & & \\
\mathcal{S}_{\texttt{constructor}} & \triangleq & \{\ s \in \mathcal{S}ym \mid \texttt{constructor} \in Attr(s)\ \} \\
\mathcal{S}'_{\texttt{constructor}} & \triangleq & \{\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \rightarrow \alpha\ [Attr \cup \{\texttt{total},\texttt{injective}\}] \\
 & & \quad \mid\ n : \forall \overrightarrow{v},\ \overrightarrow{t} \rightarrow \alpha\ [Attr] \in \mathcal{S}_{\texttt{constructor}}\ \} \\
\mathcal{S}ym' & \triangleq & (\mathcal{S}ym \setminus \mathcal{S}_{\texttt{constructor}})\ \cup\ \mathcal{S}'_{\texttt{constructor}}
\end{array}
$$

Figure 12 Formalization of the function $||\ .\ ||_{\texttt{add-attributes}}$.

Manage the fresh values. When using a 𝕂 rewriting rule, any occurrence of a fresh variable `!`X is replaced by the current value presents in the cell `GeneratedCounter` and the current value of the cell `GeneratedCounter` is replaced by a new one. For instance, the rewriting rule `rule` $\langle\ \langle\ \langle$ `foo` X $\curvearrowright$ $S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle$ `!`X $\curvearrowright$ $S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ becomes `rule` $\langle\ \langle\ \langle$ `foo` X $\curvearrowright$ $S\ \rangle_k\ \rangle_T\ \langle\ c\ \rangle_{GC}\ \rangle_{GT}$ `=>` $\langle\ \langle\ \langle\ c \curvearrowright S\ \rangle_k\ \rangle_T\ \langle\ c + 1\ \rangle_{GC}\ \rangle_{GT}$.

Add type-related symbols and extend the typing hierarchy. Implicitly, every user-defined sort is a sub-sort of `KItem`. Moreover, 𝕂 generates projection symbols and predicate symbols such as `isKResult`, as shown in Figure 13.

$$
\begin{array}{lll}
\multicolumn{3}{l}{||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\ ||_{\texttt{typing}} = (\mathcal{S}ort,\ \mathcal{R}el',\ \mathcal{S}ym',\ \mathcal{R}')} \\
\text{where} & & \\
\mathcal{R}el' & \triangleq & \mathcal{R}el\ \cup\ \{s < \texttt{KItem} \mid s \in \mathcal{S}ort \setminus \{\ \texttt{K}\ ;\ \texttt{KItem}\ \}\ \} \\
\mathcal{F}_{projection} & \triangleq & \{\ \texttt{proj}s : \texttt{K} \rightarrow s\ [\texttt{projection},\ \texttt{function}\ ] \mid s \in \mathcal{S}ort \setminus \{\ \texttt{K}\ \}\ \} \\
\mathcal{F}_{predicate} & \triangleq & \{\ \texttt{is}s : \texttt{K} \rightarrow \texttt{Bool}\ [\texttt{predicate},\ \texttt{function},\ \texttt{total}\ ] \mid s \in \mathcal{S}ort\ \} \\
\mathcal{S}ym' & \triangleq & \mathcal{S}ym\ \cup\ \mathcal{F}_{projection}\ \cup\ \mathcal{F}_{predicate} \\
\mathcal{R}_{projection} & \triangleq & \{\ s\ (\texttt{inj}_t^{\texttt{KItem}}\ X) \hookrightarrow X\ [\texttt{projection}\ ] \\
 & & \quad \mid s \in \mathcal{F}_{projection}\ \text{if}\ t\ \text{is the output type of}\ s\ \} \\
\mathcal{R}_{predicate} & \triangleq & \{\ p\ (\texttt{inj}_s^{\texttt{KItem}}\ X) \hookrightarrow \texttt{true}\ [\ ] \mid p \in \mathcal{F}_{predicate}\ \text{if}\ s\ \text{is the output sort of}\ p\ \} \\
\mathcal{R}_{pred-owise} & \triangleq & \{\ p\ X \hookrightarrow \texttt{false}\ [\texttt{owise}\ ] \mid p \in \mathcal{F}_{predicate}\ \} \\
\mathcal{R}' & \triangleq & \mathcal{R}\ \cup\ \mathcal{R}_{projection}\ \cup\ \mathcal{R}_{predicate}\ \cup\ \mathcal{R}_{pred-owise}
\end{array}
$$

Figure 13 Formalization of the function $||\ .\ ||_{\texttt{typing}}$.

Checking the coherence of the typing hierarchy. We can construct a graph where the nodes are elements of $\mathcal{S}ort$ and the edges are modelled by the elements of $\mathcal{R}el$. We reject the semantics if the graph contains at least one cycle.

Add injections. $\mathbb{K}$ adds injections (`inj`) to get full well-typed terms. This step takes into account the constraints of semantic casts (`:`), strict casts (`::`) and projection casts (`:>`).

Checking the constraints on the attribute `binder`. The first argument of a symbol with the attribute `binder` must have the sort `KVar`, which is a native $\mathbb{K}$ sort. Then, to do a substitution, it is the responsibility of each backend to correctly implement the interface proposed by $\mathbb{K}$, i.e. the file `substitution.md`.

Checking the constraints on `List` and `NeList`. It is not possible to add two or more `List` or/and `NeList` constructions at the same time to the same sort. For example, it is not allowed to write `syntax` **Exp** `::= List{Int,","} | List{Bool,","}`.

Checking the constraints on the $\mathbb{K}$ standard library. The user cannot extend the $\mathbb{K}$ standard library with `constructor` symbols, so the set $\{\ n : \forall \overrightarrow{v}, \overrightarrow{t} \to \alpha\ [Attr] \in \mathcal{S}ym \mid$ `constructor` $\in Attr$ and $\alpha \in \mathcal{S}ort_{lib}\ \}$ should be empty. That is the reason why we named a sort **MyBool** and not **Bool** in Figure 2.

Checking the constraints on rewriting rules. Finally, each rewriting rule needs to respect the BNF in Figure 14 and the sort of every condition must be boolean.

a	$::=$	$x \mid (\sigma\ a\ ...\ a)$	x is a variable
b	$::=$	$x \mid (\sigma\ b\ ...\ b) \mid (f\ b\ ...\ b)$	σ is a `constructor` symbol
c	$::=$	$f\ b\ ...\ b$	f is a `function` symbol
$rule$	$::=$	$\sigma\ a\ ...\ a \hookrightarrow b \mid \sigma\ a\ ...\ a \overset{c}{\hookrightarrow} b$	
		$\mid f\ a\ ...\ a \hookrightarrow b \mid f\ a\ ...\ a \overset{c}{\hookrightarrow} b$	

Figure 14 Constraints on rewriting rules.

We have presented the various translations carried out internally by $\mathbb{K}$. We do not claim that this list is exhaustive but it reflects our understanding of the translation from $\mathbb{K}$ to KORE. This paper formalization was elaborated by reverse engineering on KORE files as well as thanks to discussions with $\mathbb{K}$ developpers. It seems possible to print a (almost always valid) new $\mathbb{K}$ file after each transformation but this has not been implemented. So far only the following attributes have not been taken into account during the transformation: $\mathcal{A}_{modulo}$, { `priority()`, `owise` }, { `multiplicity`, `type`, `exit`, `stream` } and { `injective`, `total` }.

3.3 From $\mathbb{K}$ to Kore

We present the translation from the abstraction of $\mathbb{K}$ into KORE (Figure 15). Thanks to the obtained quadruplet, we can translate a $\mathbb{K}$ semantics into a specific MATCHING LOGIC theory named KORE. Every red keyword can be translated into a MATCHING LOGIC pattern but this translation is beyond the scope of this article. The pattern φ_{sym} can take 3 different forms corresponding to the axioms of injectivity, non-overlapping and exhaustivity of constructors.

4 From the $\mathbb{K}$ framework to the $\lambda\Pi$-calculus modulo theory

This section presents the $\lambda\Pi$-CALCULUS MODULO THEORY and DEDUKTI, a logical framework based on it. Then, we formalize the translation from $\mathbb{K}$ to the $\lambda\Pi$-CALCULUS MODULO

$\|\| (\mathcal{S}ort, \mathcal{R}el, \mathcal{S}ym, \mathcal{R}) \|\|_{\text{KORE}} =$	
`sort` $s\{\alpha_1,...,\alpha_n\}$ [ ]	for all $s \in \mathcal{S}ort \setminus \mathcal{S}ort_{lib}$
`hooked-sort` $s\{\alpha_1,...,\alpha_n\}$ [ ]	for all $s \in \mathcal{S}ort_{lib}$
`symbol` $sym\{\alpha_1,...,\alpha_n\}(\theta_1,...,\theta_m)$: θ’ [Attr]	for all $sym \in \mathcal{S}ym \setminus \mathcal{S}ym_{lib}$
`hooked-symbol` $sym\{\alpha_1,...,\alpha_n\}(\theta_1,...,\theta_m)$: θ’ [Attr]	for all $sym \in \mathcal{S}ym_{lib}$
`axiom` {R} `\exists` {R}$(x_2 : \theta_2,$ `\equals` $\{\theta_2$, R$\}(x_2 : \theta_2$, `inj` $\{\theta_1, \theta_2\}(x_1 : \theta_1)))$ [`subsort` (θ_1, θ_2)]	for all $\theta_1 < \theta_2 \in \mathcal{R}el$
`axiom` {R} `\exists` {R}$(x : \theta,$ `\equals` $\{\theta$, R$\}(x : \theta, sym\ x_1\ ...\ x_n))$ [`total`]	for all $sym \in \mathcal{S}_{\texttt{total}}$
`axiom` $\{\alpha_1,...,\alpha_n\}\ \varphi_{sym}$ [`constructor`]	for all $sym \in \mathcal{S}_{\texttt{constructor}}$
`axiom` {R} `\equals` $\{\theta$, R$\}($ $sym(sym(x_1 : \theta, x_2 : \theta), x_3 : \theta),$ $sym(x_1 : \theta, sym(x_2 : \theta, x_3 : \theta)))$ [`assoc`]	for all $sym \in \mathcal{S}_{\texttt{assoc}}$
`axiom` {R} `\equals` $\{\theta$, R$\}(sym(x_1 : \theta, x_2 : \theta),$ $sym(x_2 : \theta, x_1 : \theta))$ [`comm`]	for all $sym \in \mathcal{S}_{\texttt{comm}}$
`axiom` {R} `\equals` $\{\theta$, R$\}(sym(e, x : \theta), x : \theta)$ [`unit`]	
`axiom` {R} `\equals` $\{\theta$, R$\}(sym(x : \theta, e), x : \theta)$ [`unit`]	for all $sym \in \mathcal{S}_{\texttt{unit}}$
`axiom` {R} `\equals` $\{\theta$, R$\}(sym(x : \theta, x : \theta), x : \theta)$ [`idem`]	for all $sym \in \mathcal{S}_{\texttt{idem}}$
`axiom` {R} `\implies` {R}$(c$, `\equals` {R, R}$(l, r))$ [$Attr$]	for all $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{function}}$
`axiom` {R} `\rewrites` {R}(`\and` {R}$(c, l), r)$ [$Attr$]	for all $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R}_{\texttt{constructor}}$
where $\mathcal{S}_a \triangleq \{\ s \in \mathcal{S}ym \mid a \in Attr(s)\ \}$ and $\mathcal{R}_a \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \mid a \in Attr(head(l))\ \}$	

Figure 15 The printer to KORE.

THEORY. This translation has been implemented in a tool written in OCAML, named KAMELO, which is presented in the next section.

4.1 The $\lambda\Pi$-calculus modulo theory

The $\lambda\Pi$-CALCULUS MODULO THEORY, $\lambda\Pi{\equiv_{\mathcal{T}}}$ in short, is a logical framework, i.e. allowing to define theories, introduced by Cousineau and Dowek [10]. $\lambda\Pi{\equiv_{\mathcal{T}}}$ is an extension of the λ-calculus with dependent types and a primitive notion of computation defined thanks to rewriting rules [11]. The syntax as well as the typing rules that define the $\lambda\Pi{\equiv_{\mathcal{T}}}$ are available in Figure 16, where the typing judgment $\Gamma \vdash t : A$ means that the term t has type A with respect to the context Γ. The specific typing judgment $\Gamma \vdash A : \textbf{Type}$ indicates that A is a type under the context Γ. We also consider a signature Σ, and a set of higher-order rewriting rules $\mathcal{R}$. In this framework, for any rewriting rule $l \hookrightarrow r \in \mathcal{R}$, $FV(r) \subseteq FV(l)$ holds (where $FV(p)$ is the set of free variables of p). The use of such a rewriting rule requires that for any substitution σ, the instantiation of its left-hand side $l\sigma$ and the instantiation of its right-hand side $r\sigma$ are well-typed with the same type ($\Gamma \vdash l\sigma : A$ and $\Gamma \vdash r\sigma : A$ for a certain type A).

Syntax				
	s	:=	**Type** | **Kind**	*sort*
	t	:=	$s \mid c \mid x \mid t\ t \mid \lambda(x : t).t \mid \Pi(x : t).t$	*term*
	Γ	:=	$\emptyset \mid \Gamma, x : t$	*context*
		where	c is a constant of Σ, x is a variable	

Typing

$$\text{(sort)}\ \frac{}{\Gamma \vdash \textbf{Type} : \textbf{Kind}}$$

$$\text{(const)}\ \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash c : A}\ (c : A) \in \Sigma \qquad \text{(var)}\ \frac{\Gamma \vdash A : \textbf{Type}}{\Gamma \vdash x : A}\ (x : A) \in \Gamma$$

$$\text{(app)}\ \frac{\Gamma \vdash f : \Pi(x : A).B \qquad \Gamma \vdash a : A}{\Gamma \vdash f\ a : B\{x\backslash a\}} \qquad \text{(abs)}\ \frac{\Gamma \vdash \Pi(x : A).B : s \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda(x : A).b : \Pi(x : A).B}$$

$$\text{(prod)}\ \frac{\Gamma \vdash A : \textbf{Type} \qquad \Gamma, x : A \vdash B : s}{\Gamma \vdash \Pi(x : A).B : s} \qquad \text{(conv)}\ \frac{\Gamma \vdash t : A \qquad \Gamma \vdash B : s}{\Gamma \vdash t : B}\ A \equiv_{\beta\mathcal{R}} B$$

$$(\equiv_{reduc})\ \frac{}{\Gamma \vdash (\lambda(x : A).t)\ u \equiv t\{x\backslash u\}} \qquad (\equiv_{rule})\ \frac{\Gamma \vdash l\sigma : A \qquad \Gamma \vdash r\sigma : A}{\Gamma \vdash l\sigma \equiv r\sigma}\ l \hookrightarrow r \in \mathcal{R}$$

where $s \in \{\textbf{Type}\ ; \textbf{Kind}\}$, $B\{x\backslash a\}$ is the substitution of a for x in B, and $\equiv_{\beta\mathcal{R}}$ is the reflexive, transitive, symmetric and contextual closure of $\equiv$, generated by the rules $\equiv_{reduc}$ and $\equiv_{rule}$.

Figure 16 Syntax and typing of $\lambda\Pi{\equiv_{\mathcal{T}}}$ with a signature Σ and a rewriting system $\mathcal{R}$.

Note that in the conversion rule (conv), the equivalence relation depends not only on β-reduction but also on the rewriting system $\mathcal{R}$. Moreover, in order to have the decidability of the type-checking, the condition $A \equiv_{\beta\mathcal{R}} B$ of the rule (conv) must be decidable, which is ensured when the considered rewriting systems are convergent. Finally, contexts can contain ill-formed elements and the order of the elements does not matter. Indeed, thanks to the rule (var), only well-formed elements in the context can be used when doing a proof. This presentation has been proved equivalent to the usual presentations by Dowek [12].

4.2 Dedukti

Dedukti [4, 5] is a logical framework based on the $\lambda\Pi{\equiv}_{\mathcal{T}}$. Indeed, expressing the Calculus of Constructions in Dedukti is equivalent to defining it as a theory of the $\lambda\Pi{\equiv}_{\mathcal{T}}$. Several logics have been encoded in Dedukti, facilitating the interoperability of proofs between various formal tools [9, 22]. In this section, we only present the features available in Dedukti needed in this article.

Typing and symbols. The syntax of the $\lambda\Pi{\equiv}_{\mathcal{T}}$ is directly accessible in Dedukti: `TYPE` (**Kind** is not accessible to the user but only inferred by the system), λ (abstraction), Π (dependent product). We write $A \to B$ when the dependent product $\Pi\ (x : A), B$ is not dependent, i.e. when $x \notin FV(B)$.

The signature is defined from symbols. If the declaration of a symbol is made with the keyword `symbol` alone, the symbol is said to be *defined*, without any particular property, whereas with the additional keyword `constant`, the symbol is said to be *constant* and can not be reduced by any rewriting rule.

Rewriting rules. A Dedukti rule is written `rule` *LHS* $\hookrightarrow$ *RHS* in which the free variables are noted `$x`, `$y`, etc. As in $\mathbb{K}$, it is possible to use a wildcard (`_`) on the left-hand side when a free variable is not used in the right-hand side. Dedukti rules allow higher-order, can be non-linear and do not necessarily apply to the head of the term, but are not conditional.

4.3 Translation from abstract $\mathbb{K}$ to the $\lambda\Pi$-calculus modulo theory

Section 3 presented an abstract version of $\mathbb{K}$: any $\mathbb{K}$ file can thus be reduced to a set of sorts, subtyping relations, symbols and rewriting rules. Figure 17 presents the translation of these sets into the $\lambda\Pi$-calculus modulo theory.

$$
\begin{array}{ll}
\multicolumn{2}{l}{||\ (\mathcal{S}ort,\ \mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R})\ ||_{\lambda\Pi\equiv_{\mathcal{T}}}\ =} \\
\quad \mathtt{K} : \mathbf{Type} & \\
\quad s : \mathtt{K} & \text{for all } s \in \mathcal{S}ort \setminus \{\mathtt{K}\} \\
\quad n : \Pi(a_1 : \alpha_1), ..., \Pi(a_n : \alpha_n), \Pi(t_1 : \tau_1), ..., \Pi(t_n : \tau_n), \alpha & \text{for all } n : \forall \overrightarrow{\alpha}, \overrightarrow{\tau} \to \alpha\ [Attr] \in \mathcal{S}ym \\
\quad l \hookrightarrow r & \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional} \\
\quad ||\ \mathcal{R}_{conditional}\ ||_{\mathtt{CTRS}} & \\
\quad ||\ (\mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R}_{strategy})\ ||_{\mathtt{strategy}} & \\
\\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{unconditional} \triangleq \{\ l \hookrightarrow r \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } c = \mathtt{true}\ \}} \\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{conditional} \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } c \neq \mathtt{true} \text{ and } \{\ \mathtt{heat}, \mathtt{cool}\ \} \cap Attr = \emptyset\ \}} \\
\multicolumn{2}{l}{\qquad \text{where } \mathcal{R}_{strategy} \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{R} \text{ if } \mathtt{heat} \in Attr \text{ or } \mathtt{cool} \in Attr\ \}}
\end{array}
$$

Figure 17 From abstract $\mathbb{K}$ to the $\lambda\Pi{\equiv}_{\mathcal{T}}$.

Any sort becomes a symbol of type `K`, except the sort `K` itself, which has the type **Type**. Symbols and unconditional rules are unchanged and the set of rewriting rules obtained at the end of the translation $||\ .\ ||_{\lambda\Pi\equiv_{\mathcal{T}}}$ is $\{\ l \hookrightarrow r \mid \text{for all } l \hookrightarrow r \in \mathcal{R}_{unconditional}\ \} \cup ||\ \mathcal{R}_{conditional}\ ||_{\mathtt{CTRS}} \cup ||\ (\mathcal{R}el,\ \mathcal{S}ym,\ \mathcal{R}_{strategy})\ ||_{\mathtt{strategy}}$. The translation function $||\ .\ ||_{\mathtt{CTRS}}$ is explained in Section 4.3.1 and the translation function $||\ .\ ||_{\mathtt{strategy}}$ is explained in Section 4.3.2.

4.3.1 Translating conditional rewriting rules

In this section, we are interested in the translation of conditional rewriting rules. As conditional rewriting rules are not primitive in $\lambda\Pi{\equiv}_{\mathcal{T}}$, it is necessary to find an encoding of a conditional rewriting system ($CTRS$) into a non-conditional rewriting system (TRS).

4.3.1.1 From a CTRS to a TRS: Examples

We present two examples to illustrate the encoding of a CTRS into a TRS.

An example without owise. Consider the following system:
(1) $max\ X\ Y \overset{c}{\hookrightarrow} Y$, where $c \triangleq X < Y$
(2) $max\ X\ Y \overset{c}{\hookrightarrow} X$, where $c \triangleq X \geq Y$.
The resulting encoding is available in Figure 18 as well as an execution.

(0)	$max\ X\ Y \hookrightarrow \flat max\ X\ Y\ \flat\ \flat$	$max\ 5\ 3 \hookrightarrow_0 \flat max\ 5\ 3\ \flat\ \flat$
(1′)	$\flat max\ X\ Y\ \flat\ C \hookrightarrow \flat max\ X\ Y\ (X < Y)\ C$	$\hookrightarrow_{1'} \flat max\ 5\ 3\ (5 < 3)\ \flat$
(1″)	$\flat max\ X\ Y\ true\ C \hookrightarrow Y$	$\hookrightarrow^{*} \flat max\ 5\ 3\ false\ \flat$
(2′)	$\flat max\ X\ Y\ C\ \flat \hookrightarrow \flat max\ X\ Y\ C\ (X \geq Y)$	$\hookrightarrow_{2'} \flat max\ 5\ 3\ false\ (5 \geq 3)$
(2″)	$\flat max\ X\ Y\ C\ true \hookrightarrow X$	$\hookrightarrow^{*} \flat max\ 5\ 3\ false\ true \hookrightarrow_{2''} 5$

Figure 18 Rules generated with the variant of Viry's encoding (left) and a computation (right).

The general idea of the encoding, proposed in this section and initially proposed by Viry [24], is to add as many arguments as there are conditions for a symbol defined with conditional rules. In Figure 18, rule (0) rewrites a term whose head symbol is max into a term using the corresponding extended version of arity 4, $\flat max$ here, where all boolean arguments are $\flat$, indicating that the boolean arguments have not yet been initialised by a condition. Rules (1′) and (2′) initialize the conditions to be computed whereas rules (1″) and (2″) reduce the size of the term since one of the conditions has been evaluated to $true$. This encoding has the advantage of not fixing the order of evaluation of the conditions but increases the computation time by doubling the initial number of rules.

Contrary to Viry, we choose to extend the signature, as here with the symbol $\flat max$, rather than replacing each symbol of the signature by an equivalent symbol with a greater arity. This choice makes it possible to follow the computations of the conditions and does not force us to translate the obtained normal forms.

An example with owise. The previous example can also be written more succinctly:
$max\ X\ Y \overset{c}{\hookrightarrow} Y$, where $c \triangleq X < Y$
$max\ X\ Y \hookrightarrow X$ [owise]
To encode the attribute owise, we have two possibilities: implement an algorithm that determines the complementary condition or consider that all conditions necessarily reduce to either true or false. According to the expressiveness of the conditions that can be written in $\mathbb{K}$ (Figure 14), we add the following hypothesis: any boolean function is a total function. Under this assumption, it is not required to compute the complementary condition as we can generate a rule where every boolean argument is $false$, as shown in Figure 19. This case is formalized in 5.(c) (Figure 20).

Furthermore, $\mathbb{K}$ accepts a set of unconditional rules with at least one rule having the attribute owise. We exclude this case because we cannot model "If no other rule applies", with a boolean condition. This is equivalent to the use of the attribute priority(*nb*), with which we do not yet take into account.

$$\begin{array}{ll}
(0) & \quad max\ X\ Y \hookrightarrow \flat max\ X\ Y\ \flat \\
(1') & \flat max\ X\ Y\ \flat \hookrightarrow \flat max\ X\ Y\ (X < Y) \\
(1'') & \flat max\ X\ Y\ true \hookrightarrow Y \\
(2') & \flat max\ X\ Y\ false \hookrightarrow X
\end{array}$$

Figure 19 Rules generated with the variant of Viry's encoding, when the attribute `owise` is used.

4.3.1.2 From a CTRS to a TRS: Formalization

We present the translation, noted $||\ .\ ||_{\texttt{CTRS}}$ previously, as an algorithm in Figure 20. This translation takes as argument a set $\mathfrak{R}$ of conditional rewriting rules $l \overset{c}{\hookrightarrow} r\ [Attr]$. To avoid naming conflicts, we also assume that $\flat$ is an unused symbol name and that it does not appear in the head of any symbol name.

According to Figure 10, we need to change the definition of the *head symbol of a rule*. We note $head_{\texttt{<k>}}$ the function computing the head symbol of the cell `<k>`, for a given rule, without considering the symbols `.`, $\curvearrowright$ and `inj`. Now, we are able to define the function returning the head symbol of a rule:

$$head(l \overset{c}{\hookrightarrow} r\ [Attr]) = \begin{cases} head_{\texttt{<k>}}(l \overset{c}{\hookrightarrow} r) & \text{if the rule is a semantical rule and } \texttt{anywhere} \notin Attr \\ f & \text{otherwise, where } l \triangleq f\ a\ ...\ a \text{ (Figure 14)} \end{cases}$$

We also define the set of rules noted $\mathcal{C}_\sigma$, in which the rules share the same head symbol σ, that is $\mathcal{C}_\sigma \triangleq \{\ l \overset{c}{\hookrightarrow} r\ [Attr] \mid head\ (l \overset{c}{\hookrightarrow} r\ [Attr]) = \sigma\ \}$. We assume that, if there is a rule with the attribute `owise` in $\mathcal{C}_\sigma$ for a given σ, all the other rules in $\mathcal{C}_\sigma$ must have a condition.

After calculating each set $\mathcal{C}_\sigma$ from $\mathfrak{R}$, we run the algorithm presented in Figure 20, for each $\mathcal{C}_\sigma$, where X is the number of conditional rules in $\mathcal{C}_\sigma$.

1. If $X = 0$, $\mathcal{C}_\sigma$ is unchanged and the algorithm stops. Otherwise, initialize i to 0 and go to 2.
2. Generate the most general left-hand side for a given symbol σ, noted $mglhs_\sigma$.
3. Generate the extended symbol $\flat\sigma$ of type $T_1 \to ... \to T_{n-1} \to \flat\texttt{Bool} \to ... \to \flat\texttt{Bool} \to T_n$, with X argument(s) of type $\flat\texttt{Bool}$, where $\flat\texttt{Bool} = \texttt{Bool} \cup \{\flat\}$, and σ of type $T_1 \times ... \times T_{n-1} \to T_n$.
4. Generate the substitution rule: $mglhs_\sigma \hookrightarrow mglhs_\sigma[\ update_{same}(\sigma, \flat\sigma, \flat)\ ]_\sigma$
5. For each rule $l \overset{c}{\hookrightarrow} r\ [Attr] \in \mathcal{C}_\sigma$:
   a. If $c \neq \texttt{true}$ and `owise` $\notin Attr$:
      - Increment i by 1
      - Generate the initialization rule:
        $l\ [\ update_{diff}(\sigma, \flat\sigma, \flat, i, \_)\ ]_\sigma \hookrightarrow l\ [\ update_{diff}(\sigma, \flat\sigma, c, i, \_)\ ]_\sigma$
      - Generate the reduction rule: $l\ [\ update_{diff}(\sigma, \flat\sigma, \texttt{true}, i, \_)\ ]_\sigma \hookrightarrow r$
   b. If $c = \texttt{true}$ and `owise` $\notin Attr$:
      - Generate the reduction rule: $l\ [\ update_{same}(\sigma, \flat\sigma, \_)\ ]_\sigma \hookrightarrow r$
   c. If $c = \texttt{true}$ and `owise` $\in Attr$:
      - Generate the reduction rule: $l\ [\ update_{same}(\sigma, \flat\sigma, \texttt{false})\ ]_\sigma \hookrightarrow r$

where:
* $t_1[t_2]_\sigma$ means that we substitute t_2 for the subterm with the head symbol σ in t_1.
* $arg_i(t)$ corresponds to the i-th argument of t and $arity(t)$ to the number of arguments of t.
* $mglhs_\sigma \triangleq (\text{mgconf } init\text{-}config)\ [\ [\texttt{k}]_{\texttt{K}}(x) \setminus [\texttt{k}]_{\texttt{K}}((\texttt{inj}^{\texttt{KItem}}_{RetType(\sigma)}\ \sigma\ f_1\ ...\ f_{arity(\sigma)}) \curvearrowright L)\ ]$,
where $init\text{-}config$ is the initial configuration, and f_i and L are fresh variables.

* $update_{diff}(\sigma, \flat\sigma, s_1, i, s_2) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}\ with\ x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \leq j \leq arity(\sigma) \\ s_1 & \text{if } j = arity(\sigma) + i \\ s_2 & \text{otherwise} \end{cases}$

* $update_{same}(\sigma, \flat\sigma, s) = \flat\sigma\ x_1\ ...\ x_{arity(\sigma)+X}\ with\ x_j = \begin{cases} arg_j(\sigma) & \text{if } 1 \leq j \leq arity(\sigma) \\ s & \text{otherwise} \end{cases}$

Figure 20 Variant of Viry's encoding.

4.3.2 Translating evaluation strategies

As we saw in Section 3.2, some conditional rewriting rules can be generated during the translation of $\mathbb{K}$ to Kore, as is the case for the evaluation strategies defined by the attributes `strict` and `seqstrict`. The rewriting rules generated by these attributes require the translation of the K computations, i.e. the symbols $.$ and $\curvearrowright$, the freezers but also the predicate `isKResult`. However, these conditional rewriting rules are part of a known case where Viry's encoding is not confluent, notably because the order of application of some rewriting rules modifies the result of the condition, which can stop the computation. Figure 21 shows the translation of the rules of Figure 3 with Viry's encoding (on the right) and an example of a valid but stuck execution[1] (on the left).

1	$E1 \;\&\&\; E2 \curvearrowright C \hookrightarrow$	$(true \;\&\&\; true) \;\&\&\; false \curvearrowright .$
	$\flat\&\&\; E1\; E2\, \flat \curvearrowright C$	$\hookrightarrow_1 \flat\&\&\; (true \;\&\&\; true)\; false\, \flat \curvearrowright .$
2	$\flat\&\&\; E1\; E2\, \flat \curvearrowright C \hookrightarrow$	$\hookrightarrow_2 \flat\&\&\; (true \;\&\&\; true)$
	$\flat\&\&\; E1\; E2\; (not(\mathtt{isKResult}\; E1)) \curvearrowright C$	$\quad false$
3	$\flat\&\&\; E1\; E2\; true \curvearrowright C \hookrightarrow E1 \curvearrowright (\text{❄}^1_{\&\&}\; \mathrm{E2}) \curvearrowright C$	$\quad (not(\mathtt{isKResult}\; (true \;\&\&\; true))) \curvearrowright .$
4	$E1 \curvearrowright (\text{❄}^1_{\&\&}\; \mathrm{E2}) \curvearrowright C \hookrightarrow$	$\hookrightarrow^* \; \flat\&\&\; (true \;\&\&\; true)\; false\; true \curvearrowright .$
	$(\flat\text{❄}^1_{\&\&}\; \mathrm{E1}\; \mathrm{E2}\, \flat) \curvearrowright C$	$\hookrightarrow_3 (true \;\&\&\; true) \;\curvearrowright (\text{❄}^1_{\&\&}\; false) \curvearrowright .$
5	$(\flat\text{❄}^1_{\&\&}\; \mathrm{E1}\; \mathrm{E2}\, \flat) \curvearrowright C \hookrightarrow$	$\hookrightarrow_4 (\flat\text{❄}^1_{\&\&}\; (true \;\&\&\; true)\; false\, \flat) \curvearrowright .$
	$(\flat\text{❄}^1_{\&\&}\; \mathrm{E1}\; \mathrm{E2}\; (\mathtt{isKResult}\; \mathrm{E1})) \curvearrowright C$	$\hookrightarrow_5 (\flat\text{❄}^1_{\&\&}\; (true \;\&\&\; true)$
6	$(\flat\text{❄}^1_{\&\&}\; \mathrm{E1}\; \mathrm{E2}\; true) \curvearrowright C \hookrightarrow E1 \;\&\&\; E2 \curvearrowright C$	$\quad false$
7	$true \;\;\&\&\; B \curvearrowright C \hookrightarrow B \curvearrowright C$	$\quad (\mathtt{isKResult}\; (true \;\&\&\; true))) \curvearrowright .$
8	$false \;\&\&\; \_ \curvearrowright C \hookrightarrow false \curvearrowright C$	$\hookrightarrow^* \; (\flat\text{❄}^1_{\&\&}\; (true \;\&\&\; true)\; false\; false) \curvearrowright .$

Figure 21 Rules generated with previous encoding (left) and a stuck execution (right).

Intuitively, these rules are used to ensure that E_1 is of a specific sort in order to allow or not its evaluation. The idea of our new encoding is to specialize some terms of the rules, i.e. to refine the pattern-matching in order to ensure the desired type. For example, rule 2. in Figure 3 becomes $\langle\; (\mathtt{inj}^{\mathtt{KItem}}_{\mathtt{Bool}}\; E_1) \curvearrowright (\text{❄}^1_{\&\&}\; E_2) \curvearrowright S\; \rangle_k \hookrightarrow \langle\; (\mathtt{inj}^{BExp}_{\mathtt{Bool}}\; E_1) \;\&\&\; E_2 \curvearrowright S\; \rangle_k$, where we force E_1 to have the sort `Bool`. Thus this rule can be used only if the term E_1 is a fully computed Boolean expression. Morevoer, rule 1. in Figure 3 becomes the single rule $\langle\; (X_1 \;\&\&\; X_2) \;\&\&\; E_2 \curvearrowright S\; \rangle_k \hookrightarrow \langle\; (X_1 \;\&\&\; X_2) \curvearrowright (\text{❄}^1_{\&\&}\; E_2) \curvearrowright S\; \rangle_k$ because there is only one constructor associated to **BExp** and no `token` symbol. Symbols with the attribute `function` or `macro` are not considered, because they are not allowed in the left-hand side of a rule, as well as symbols with the attribute `bracket`, because these ones disappear during the compilation process of $\mathbb{K}$. The full formalization is available in Figure 22.

$$
\begin{array}{lcl}
\multicolumn{3}{l}{||\; (\mathcal{R}el,\, \mathcal{S}ym,\, \mathcal{R})\; ||_{\mathtt{strategy}} = \mathcal{R}'} \\
\multicolumn{3}{l}{\quad \text{where}} \\
\mathcal{S}ub & \triangleq & \{\; s \mid s < \mathtt{KResult} \in \mathcal{R}el\; \} \\
\mathcal{R}_{\mathtt{cool}} & \triangleq & \{\; r \in \mathcal{R} \mid \mathtt{cool} \in Attr(r)\; \} \\
\mathcal{R}'_{\mathtt{cool}} & \triangleq & \{\; (l \hookrightarrow r)\; [\; x \setminus \mathtt{inj}^{\mathtt{KItem}}_{s}\; x\;] \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\mathtt{cool}} \text{ where } c \triangleq (\mathtt{isKResult}\; x),\, s \in \mathcal{S}ub\; \} \\
\mathcal{R}_{\mathtt{heat}} & \triangleq & \{\; r \in \mathcal{R} \mid \mathtt{heat} \in Attr(r)\; \} \\
\mathcal{S}^{s_2}_{s_1} & \triangleq & \{\; \mathtt{inj}^{s_2}_{s}\; f \mid \text{where } f \text{ is a fresh variable and } s \in (\{\; s \mid s < s_1 \in \mathcal{R}el\} \setminus \mathcal{S}ub)\; \} \\
\mathcal{P}^{s_2}_{s_1} & \triangleq & \{\; \mathtt{inj}^{s_2}_{s_1}(n\, \overrightarrow{f}) \mid \text{where } \overrightarrow{f} \text{ are fresh variables and } n : \forall \overrightarrow{v},\, \overrightarrow{t} \to \alpha\; [Attr] \in \mathcal{S}ym \\
& & \qquad \text{if } \alpha = s_1 \text{ and } \{\; \mathtt{constructor},\, \mathtt{token}\; \} \cap Attr \neq \emptyset\; \} \\
\mathcal{R}'_{\mathtt{heat}} & \triangleq & \{\; (l \hookrightarrow r)\; [\; x_1,\, \dots\, ,\, x_k \setminus \mathtt{inj}^{\mathtt{KItem}}_{s_1}\; x_1,\, \dots\, ,\, \mathtt{inj}^{\mathtt{KItem}}_{s_k}\; x_k\;]\; [\; \mathtt{inj}^{s_2}_{s_1}\; x \setminus t\;] \\
& & \qquad \mid l \overset{c}{\hookrightarrow} r \in \mathcal{R}_{\mathtt{heat}} \text{ and } (s_1, \dots, s_k) \in \mathcal{S}ub^k \text{ and } t \in (\mathcal{S}^{s_2}_{s_1}\; \cup\; \mathcal{P}^{s_2}_{s_1}), \\
& & \qquad\quad \text{where } c \triangleq (\mathtt{isKResult}\; x_1 \wedge \dots \wedge \mathtt{isKResult}\; x_k \wedge \neg\, (\mathtt{isKResult}\; \mathtt{inj}^{s_2}_{s_1}\; x)\; \} \\
\mathcal{R}' & \triangleq & \mathcal{R} \setminus (\mathcal{R}_{\mathtt{heat}}\; \cup\; \mathcal{R}_{\mathtt{cool}}) \cup (\mathcal{R}'_{\mathtt{heat}} \cup \mathcal{R}'_{\mathtt{cool}})
\end{array}
$$

Figure 22 Specialization of the evaluation strategy rules.

[1] It is also possible to obtain `false`.

4.3.3 Semantics preservation

The soundness of the translation is not formally proved in this article. Informally, our translation seeks to ensure that the program executed in the $\mathbb{K}$ framework and the program executed in Dedukti have the same behaviour. If the language described is deterministic, $\mathbb{K}$ and Dedukti compute the same value or give the same final state. If the language is non-deterministic, $\mathbb{K}$ allows to obtain all possible final configurations. In Dedukti, it is only possible to obtain one final configuration, because the algorithm is deterministic.

As previously, we assume that every condition is reducible into `false` or `true`. We also assume that the $\mathbb{K}$ semantics does not use the following attributes: no cell of the configuration has one of the attributes `multiplicity`, `stream`, `type`, `exit`, no evaluation strategy based on `result` or `hybrid`, and no rewriting rule has the attribute `priority()`, `unboundVariables`, `assoc`, `comm`, `unit` or `idem`. Lastly, we assume that, for a given symbol, among the associated evaluation rules, only one rule has the attribute `owise` and in this case, other rules must have a condition.

The two following parts present the soundness and completeness statements, thanks to the function $|\ .\ |$ which is translated a $\mathbb{K}$ term into a Dedukti one by induction on Term($\mathbb{K}$):

- $|\ sym\ x_1\ \dots\ x_n\ | \triangleq sym\ |\ x_1\ |\ \dots\ |\ x_n\ |$, if $sym \in \Sigma_{\text{Dedukti}}$,
- $|\ x\ | \triangleq x$, where x is a variable

4.3.3.1 Soundness

The next conjecture helps to assert that any derivation in $\mathbb{K}$ is also a derivation in Dedukti.

▶ **Conjecture 1.** *For any $\mathbb{K}$ rewriting step $l \hookrightarrow r$, there is a DK derivation $|\ l\ | \hookrightarrow^* |\ r\ |$.*

▶ **Corollary** (From $\mathbb{K}$ to Dedukti)**.** *For every derivation $l \hookrightarrow^* r$ in $\mathbb{K}$, there is a derivation $|\ l\ | \hookrightarrow^* |\ r\ |$ in Dedukti.*

4.3.3.2 Completeness

We note $\mathcal{F}lat$ the set of every term starting with $\flat$.
We note $\mathcal{G}host$ the set of every term having at least one symbol in $\mathcal{F}lat$.

▶ **Lemma 1.** *$|\ .\ | : Term(\mathbb{K}) \to Term(\text{Dedukti}) \setminus \mathcal{G}host$ is a bijection.*

We define the *translation function* $||\ .\ ||_{\texttt{K2DK}}$: Term($\mathbb{K}$) → Term(Dedukti) such that $||\ t\ ||_{\texttt{K2DK}} = |\ t\ |$ and the *detranslation function* $||\ .\ ||_{\texttt{DK2K}}$: Term(Dedukti) → Term($\mathbb{K}$) such that $||\ t\ ||_{\texttt{DK2K}} = \begin{cases} |\ t\ |^{-1} & \text{if } t \notin \mathcal{G}host \\ ||\ t\ ||_{\texttt{forget}} & \text{if } t \in \mathcal{G}host \end{cases}.$

The function `forget` is defined inductively on Term(Dedukti):

- $||\ sym\ x_1\ \dots\ x_n\ ||_{\texttt{forget}} \triangleq sym\ ||\ x_1\ ||_{\texttt{forget}} \dots ||\ x_n\ ||_{\texttt{forget}}$, if $sym \notin \mathcal{F}lat$
- $||\ \flat sym\ x_1\ \dots\ x_n\ ||_{\texttt{forget}} \triangleq sym\ ||\ x_1\ ||_{\texttt{forget}} \dots ||\ x_i\ ||_{\texttt{forget}}$,
  if $\flat sym \in \mathcal{F}lat$, where $x_{i+1}\ \dots\ x_n$ are conditions
- $||\ x\ ||_{\texttt{forget}} \triangleq x$, where x is a variable

The following conjecture states that any derivation in Dedukti is also a derivation in $\mathbb{K}$, except if the derivation begins or ends with a term generated by the Viry encoding.

▶ **Conjecture 2** (From Dedukti to $\mathbb{K}$)**.** *For every derivation $l \hookrightarrow^* r$ in Dedukti, there is a derivation $||\ l\ ||_{DK2K} \hookrightarrow^* ||\ r\ ||_{DK2K}$ in $\mathbb{K}$ if $l \notin \mathcal{G}host$ or $r \notin \mathcal{G}host$.*

▶ **Corollary** (Preservation of confluence)**.** *If the rewriting system $\mathcal{R}$ written in $\mathbb{K}$ is confluent, then the translation of the rewriting system $\mathcal{R}$ in Dedukti is confluent.*

▶ **Corollary** (Preservation of termination). *If the rewriting system $\mathcal{R}$ written in $\mathbb{K}$ is terminating, then the translation of the rewriting system $\mathcal{R}$ in Dedukti is terminating.*

5 Implementation and examples

This section focuses on the implementation of the translations presented in the previous section, i.e. on the tool KaMeLo [1] which allows to translate Kore into Dedukti.

5.1 KaMeLo in a nutshell

In practice, the formalizations presented in Section 3.2 as well as the printer to Kore (Figure 15) correspond to the command `kompile` implemented by the $\mathbb{K}$ team. Like the command `krun`, also implemented by the $\mathbb{K}$ team, KaMeLo allows programs translated into Kore to be executed in Dedukti thanks to the $\mathbb{K}$ semantics translated into Kore. KaMeLo implements the translation formalized in Figure 20 and Figure 22.

Moreover, it is the responsibility of each backend to implement the $\mathbb{K}$ standard library and to support the appropriate attributes. The backend KaMeLo does not support the attributes `multiplicity`, `stream`, `type`, `exit`, `result`, `hybrid`, `priority()`, `unboundVariables`, `assoc`, `comm`, `unit` and `idem`. The implementation of the $\mathbb{K}$ standard library in Dedukti is available on `https://gitlab.com/semantiko/DK-BiblioteKo`.

5.2 KaMeLo in action

From a semantics of 84 lines of a simple while-language similar to IMP in [25], it is possible to obtain a Kore file of 4 130 lines (18 sorts, 5 hooked sorts, 102 symbols, 78 hooked symbols and 552 axioms). However, in order to execute a program, the axioms with the attributes `subsort`, `total`, `constructor`, `assoc`, `comm`, `unit` or `idem` do not need to be translated. The translation of this semantics in Dedukti has 723 lines (122 symbols and 122 rewriting rules). To execute the following program computing the GCD of x and y

```
decl x, y ; x = 20 ; y = 15 ;
while not( (y <= x) and (x <= y) ) do
  { if y < x then x = x - y ; else y = y - x ; }
```

the command `$ krun --depth 0 --output kore GCD.imp` allows to translate the program in Kore. After translating it into Dedukti, the result is: `<generatedTop> (<T> (<k> .) (<env> (inj y ↦ inj 5) ; (inj x ↦ inj 5))) (<generatedCounter> 0);`. The source code of KaMeLo [1] is joined by some tests as the one presented here.

6 Conclusion

This article presents a paper formalization of the translation from $\mathbb{K}$ into Kore and, a paper formalization and an automatic tool, called KaMeLo, from Kore to Dedukti, in order to execute programs in Dedukti. There has already been a translation of a programming language in Dedukti [8, 9], but this is the first time a semantical framework has been translated into Dedukti.

This work needs to be extended to take into account the attributes `priority()`/`owise`, `multiplicity`/`type` and `result`/`hybrid`. The attributes `assoc`, `comm`, `unit`, `idem` and `unboundVariables` can theoretically not be translated in the general case.

The verification of proof objects generated by the KProver as well as the encoding of the theoretical foundations of $\mathbb{K}$ into those of Dedukti, are not in the scope of this article and will be the subject of future work. The translation presented here is nevertheless necessary to run a program and will be reused for proof checking.

References

1 GitLab of KaMeLo. URL: `https://gitlab.com/semantiko/kamelo`.

2 Website of $\mathbb{K}$. URL: `https://kframework.org/`.

3 Website of Sail. URL: `https://www.cl.cam.ac.uk/~pes20/sail/`.

4 Ali Assaf, Guillaume Burel, Raphal Cauderlier, David Delahaye, Gilles Dowek, Catherine Dubois, Frédéric Gilbert, Pierre Halmagrand, Olivier Hermant, and Ronan Saillard. Expressing theories in the λΠ-calculus modulo theory and in the Dedukti system. In *TYPES: Types for Proofs and Programs*, Novi SAd, Serbia, May 2016. URL: `https://hal-mines-paristech.archives-ouvertes.fr/hal-01441751`.

5 Frédéric Blanqui, Gilles Dowek, Émilie Grienenberger, Gabriel Hondet, and François Thiré. Some axioms for mathematics. In Naoki Kobayashi, editor, *6th International Conference on Formal Structures for Computation and Deduction, FSCD 2021, July 17-24, 2021, Buenos Aires, Argentina (Virtual Conference)*, volume 195 of *LIPIcs*, pages 20:1–20:19. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. `doi:10.4230/LIPIcs.FSCD.2021.20`.

6 Denis Bogdănaş and Grigore Roşu. K-Java: A Complete Semantics of Java. In *Proceedings of the 42nd Symposium on Principles of Programming Languages (POPL'15)*, pages 445–456. ACM, January 2015. `doi:10.1145/2676726.2676982`.

7 Patrick Borras, Dominique Clément, Th. Despeyroux, Janet Incerpi, Gilles Kahn, Bernard Lang, and V. Pascual. CENTAUR: The System. In Peter B. Henderson, editor, *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, Boston, Massachusetts, USA, November 28-30, 1988*, pages 14–24. ACM, 1988. `doi:10.1145/64135.65005`.

8 Raphaël Cauderlier and Catherine Dubois. ML Pattern-Matching, Recursion, and Rewriting: From FoCaLiZe to Dedukti. In *ICTAC 2016 - 13th International Colloquium on Theoretical Aspects of Computing*, volume 9965 of *LNCS*, pages 459–468, Taipei, Taiwan, October 2016. `doi:10.1007/978-3-319-46750-4_26`.

9 Raphaël Cauderlier and Catherine Dubois. FoCaLiZe and Dedukti to the rescue for proof interoperability. In Mauricio Ayala-Rincón and César A. Muñoz, editors, *ITP 2017: International Conference on Interactive Theorem Proving*, page 532, Brasília, Brazil, September 2017. `doi:10.1007/978-3-319-66107-0_9`.

10 D. Cousineau and Gilles Dowek. Embedding Pure Type Systems in the Lambda-Pi-Calculus Modulo. In *TLCA*, 2007.

11 Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In *Handbook of Theoretical Computer Science, Volume B: Formal Models and Semantics*, pages 243–320, 1990.

12 Gilles Dowek. Interacting Safely with an Unsafe Environment. *CoRR*, abs/2107.07662, 2021. `arXiv:2107.07662`, `doi:10.4204/EPTCS.337.3`.

13 Chris Hathhorn, Chucky Ellison, and Grigore Roşu. Defining the Undefinedness of C. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 336–345. ACM, June 2015. `doi:10.1145/2813885.2737979`.

14 Liyi Li and Elsa L. Gunter. IsaK: A Complete Semantics of $\mathbb{K}$. Technical report, University of Illinois at Urbana-Champaign, June 2018. URL: `https://hdl.handle.net/2142/100116`.

15 Liyi Li and Elsa L. Gunter. IsaK-static: A complete static semantics of $\mathbb{K}$. In *Formal Aspects of Component Software - 15th International Conference, FACS 2018, Proceedings*, pages 196–215. Springer-Verlag Berlin Heidelberg, 2018. `doi:10.1007/978-3-030-02146-7_10`.

16 Liyi Li and Elsa L. Gunter. A Complete Semantics of 𝕂 and Its Translation to Isabelle. In Antonio Cerone and Peter Csaba Ölveczky, editors, *Theoretical Aspects of Computing – ICTAC 2021*, pages 152–171, Cham, 2021. Springer International Publishing.

17 Dominic Mulligan, Scott Owens, Kathryn Gray, Tom Ridge, and Peter Sewell. Lem: Reusable Engineering of Real-world Semantics. *ACM SIGPLAN Notices*, 49, August 2014. `doi:10.1145/2628136.2628143`.

18 Daejun Park, Andrei Ştefănescu, and Grigore Roşu. KJS: A Complete Formal Semantics of JavaScript. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*, pages 346–356. ACM, June 2015. `doi:10.1145/2737924.2737991`.

19 Grigore Roșu and Traian Florin Șerbănută. An overview of the 𝕂 semantic framework. *The Journal of Logic and Algebraic Programming*, 79(6):397–434, August 2010. `doi:10.1016/j.jlap.2010.03.012`.

20 Peter Sewell, Francesco Zappa Nardelli, Scott Owens, Gilles Peskine, Thomas Ridge, Susmit Sarkar, and Rok Strniša. Ott: Effective tool support for the working semanticist. *Journal of Functional Programming*, 20(1):71–122, 2010. `doi:10.1017/S0956796809990293`.

21 Andrei Stefănescu, Daejun Park, Shijiao Yuwen, Yilong Li, and Grigore Roşu. Semantics-based program verifiers for all languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 74–91, Amsterdam Netherlands, October 2016. ACM. `doi:10.1145/2983990.2984027`.

22 François Thiré. Sharing a library between proof assistants: Reaching out to the HOL family. *Electronic Proceedings in Theoretical Computer Science*, 274:57–71, July 2018. `doi:10.4204/eptcs.274.5`.

23 M. G. J. van den Brand, A. van Deursen, J. Heering, H. A. de Jong, M. de Jonge, T. Kuipers, P. Klint, L. Moonen, P. A. Olivier, J. Scheerder, J. J. Vinju, E. Visser, and J. Visser. The Asf+Sdf Meta-environment: A Component-Based Language Development Environment. In Reinhard Wilhelm, editor, *Compiler Construction*, pages 365–370, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg. `doi:10.1007/3-540-45306-7_26`.

24 Patrick Viry. Elimination of Conditions. *Journal of Symbolic Computation*, 28(3):381–401, 1999. `doi:10.1006/jsco.1999.0288`.

25 Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

26 A.K. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, November 1994. `doi:10.1006/inco.1994.1093`.

Type Theory with Explicit Universe Polymorphism

Marc Bezem
University of Bergen, Norway

Thierry Coquand
University of Gothenburg, Sweden

Peter Dybjer
Chalmers University of Technology, Gothenburg, Sweden

Martín Escardó
University of Birmingham, UK

Abstract

The aim of this paper is to refine and extend proposals by Sozeau and Tabareau and by Voevodsky for universe polymorphism in type theory. In those systems judgments can depend on explicit constraints between universe levels. We here present a system where we also have products indexed by universe levels and by constraints. Our theory has judgments for internal universe levels, built up from level variables by a successor operation and a binary supremum operation, and also judgments for equality of universe levels.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, universes in type theory, universe polymorphism, level-indexed products, constraint-indexed products

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.13

Acknowledgements The authors are grateful to the anonymous referees for useful feedback, and to Matthieu Sozeau for an update on the current state of universe polymorphism in Coq. We acknowledge the support of the Centre for Advanced Study (CAS) at the Norwegian Academy of Science and Letters in Oslo, Norway, which funded and hosted the research project Homotopy Type Theory and Univalent Foundations during the academic year 2018/19.

1 Introduction

The system of simple type theory, as introduced by Church [9], is elegant and forms the basis of several proof assistants. However, it has some unnatural limitations: it is not possible in this system to talk about an arbitrary type or about an arbitrary structure. For example, it is not possible to form the collection of all groups as needed in category theory. In order to address these limitations, Martin-Löf [22, 21] introduced a system with a type V of all types. A function $A \to V$ in this system can then be seen as a family of types over a given type A. It is natural in such a system to refine the operations exponential and cartesian product in simple type theory to operations of dependent products and sums. After the discovery of Girard's paradox [16], Martin-Löf [23] introduced a distinction between *small* and *large* types, similar to the distinction introduced in category theory between large and small sets, and the type V became the (large) type of small types. The name "universe" for such a type was chosen in analogy with the notion of universe introduced by Grothendieck to represent category theory in set theory.

Later, Martin-Löf [24] introduced a countable sequence of universes

$$\mathsf{U}_0 : \mathsf{U}_1 : \mathsf{U}_2 : \cdots$$

We refer to the indices $0, 1, 2, \ldots$ as *universe levels*.

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 13; pp. 13:1–13:16
Leibniz International Proceedings in Informatics
Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Before the advent of univalent foundations, most type theorists expected only the first few universe levels to be relevant in practical formalisations. One thought that it might be feasible for a user of type theory to explicitly assign universe levels to their types and then simply add updated versions of earlier definitions when they were needed at different levels. However, the number of copies of definitions does not only grow with the level, but also with the number of type arguments in the definition of a type former. (The latter growth can be exponential!)

To deal with this, Huet [20] introduced a specific form of universe polymorphism that allowed the use of $\mathsf{U} : \mathsf{U}$ on the condition that each occurrence of U can be disambiguated as U_i in a consistent way. This approach has been followed by Harper and Pollack [18] and in Coq [35]. These approaches to *implicit* universe polymorphism are, however, problematic with respect to modularity. As pointed out in [11, 28]: one can prove $A \to B$ in one file, and $B \to C$ in another file, while $A \to C$ is not valid.

Leaving universe levels implicit also causes practical problems, since universe level disambiguation can be a costly operation, slowing down type-checking significantly. Moreover, so-called universe inconsistencies can be hard to explain to the user.

In order to cope with these issues, Courant [11] introduced explicit universe levels, with a supremum operation (see also Herbelin [19]). Explicit universe levels are also present in Agda [32] and Lean [12, 7]. However, whereas Courant has universe level *judgments*, Agda has a *type* of universe levels, and hence supports the formation of level-indexed products.

With the advent of Voevodsky's univalent foundations, the need for universe polymorphism has only increased. One often wants to prove theorems uniformly for arbitrary universes. These theorems may depend on several universes and there may be constraints on the level of these universes. In response to this Voevodsky [39] and Sozeau and Tabareau [30] proposed type theories parameterized by (arbitrary but fixed) universe levels and constraints.

The *univalence axiom* states that for any two types X, Y the canonical map

$$\mathsf{idtoeq}_{X,Y} : (X = Y) \to (X \simeq Y)$$

is an equivalence. Formally, the univalence axiom is an axiom scheme which is added to Martin-Löf type theory. If we work in Martin-Löf type theory with a countable tower of universes, each type is a member of some universe U_n. Such a universe U_n is *univalent* provided for all $X, Y : \mathsf{U}_n$ the canonical map $\mathsf{idtoeq}_{X,Y}$ is an equivalence. Let UA_n be the type expressing the univalence of U_n, and let $\mathsf{ua}_n : \mathsf{UA}_n$ for $n = 0, 1, \ldots$ be a sequence of constants postulating the respective instances of the univalence axiom. We note that $X = Y : \mathsf{U}_{n+1}$ and $X \simeq Y : \mathsf{U}_n$ and hence $\mathsf{UA}_n : \mathsf{U}_{n+1}$. We can express the universe polymorphism of these judgments internally in all of the above-mentioned systems by quantifying over universe levels, irrespective of having universe level judgments or a type of universe levels.

To be explicit about universes can be important, as shown by Waterhouse [40, 8], who gives an example of a large presheaf with no associated sheaf. A second example is the fact that the embedding $\mathsf{Group}(\mathsf{U}_n) \to \mathsf{Group}(\mathsf{U}_{n+1})$ of the type of groups in a universe U_n into that of the next universe U_{n+1} is not an equivalence. That is, there are more groups in the next universe [5].

We remark that universes are even more important in a predicative framework than in an impredicative one, for uniform proofs and modularity. Consider for example the formalisation of real numbers as Dedekind cuts, or domain elements as filters of formal neighbourhoods. Both belong to U_1 since they are properties of elements in U_0. However, even in a system using an impredicative universe of propositions, such as the ones in [20, 12], there is a need for definitions parametric in universe levels.

Terminology

Following Cardelli [6], we distinguish between *implicit* and *explicit* polymorphism:

> Parametric polymorphism is explicit when parametrization is obtained by explicit type parameters in procedure headings, and corresponding explicit applications of type arguments when procedures are called ... Parametric polymorphism is called implicit when the above type parameters and type applications are not admitted, but types can contain type variables which are unknown, yet to be determined, types.

Motivation

Many substantial Agda developments make essential use of *explicit* universe polymorphism with successor and finite suprema. Examples include the Agda standard library [34], the cubical Agda library [36], 1Lab [33], the Agda-HoTT library [37], agda-unimath [27], TypeTopology [14], HoTT-UF-in-Agda [13] (Midlands Graduate School 2019 lecture notes).

The original motivation for this work was to formalise the type theory of Agda, including explicit universe polymorphism. In doing that, we found ourselves modifying Agda's treatment of universes as follows:

- We have universe level *judgments*, like Courant [11], instead of a *type* of universe levels, like Agda.
- We add the possibility of expressing *explicit* universe level constraints. This is not only more general but also arguably gives a more natural way of expressing types involving universes.
- We do not require a first universe level zero, so that every definition that involves universes is polymorphic.
- We include a Type judgment, which does not refer to universes, as in Martin-Löf [25].

Our resulting type theory is orthogonal to the presence or absence of cumulativity. In the body of the paper, we treat universes à la Tarski, but we also give an appendix with a version à la Russell.

We have checked that the lecture notes [13] on HoTT/UF, which include 9620 lines of Agda without comments, can be rewritten without universe level zero. We believe, based on what we learned from this experiment, that the above Agda developments could also be rewritten in this way. Experience with these Agda developments suggest that a *type* for levels in Agda could be replaced by level judgments in practice. The fact that levels form a type in Agda automatically allows for nested universal quantification over levels, which we instead add explicitly to our type theory.

Summary of main contributions

Like Courant, we present a type theory with universe levels and universe level equations as *judgments*. Moreover, we don't restrict the levels to be natural numbers. Instead we just assume that they form a sup-semilattice with an inflationary endomorphism. In this way all levels are built up from level variables by a successor operation and a binary supremum operation. Unlike most other systems, we do not have a level constant 0 for the first universe level. Thus all types involving universes depend on level variables; they are *universe polymorphic*.

Furthermore, we make the polymorphism fully explicit in the sense of Cardelli by adding *level-indexed products*. In this way we regain some of the expressivity Agda gets from having a *type* Level of universe levels. Finally, we present a type theory with constraints as *judgments* similar to the ones by Sozeau and Tabareau [30] and Voevodsky [39] but extended with *constraint-indexed products*.

Plan

In Section 2 we display rules for a basic version of dependent type theory with Π, Σ, N, and an identity type former Id.

In Section 3 we explain how to add an externally indexed sequence of universes $\mathsf{U}_n, \mathsf{T}_n$ ($n \in \mathbb{N}$) à la Tarski, without cumulativity rules. In Appendix A we present a system with cumulativity, and in Appendix B we present a system à la Russell.

In Section 4 we introduce a notion of universe level, and let judgments depend not only on a context of ordinary variables, but also on level variables $\alpha, \dots, \beta$. This gives rise to a type theory with level polymorphism, which we call "ML-style" as long as do not bind level variables. We then extend this theory with level-indexed products of types $[\alpha]A$ and corresponding abstractions $\langle\alpha\rangle A$ to give full level polymorphism.

In Section 5 we extend the type theory in Section 4 with constraints (lists of equations between level expressions). Constraints can now appear as assumptions in hypothetical judgments. Moreover, we add constraint-indexed products of types $[\psi]A$ and corresponding abstractions $\langle\psi\rangle A$. This goes beyond the systems of Sozeau and Tabareau [30] and Voevodsky [39]. In Section 6 we compare our type theory with Voevodsky's and Sozeau-Tabareau's and briefly discuss some other approaches. Finally, in Section 7 we outline future work.

2 Rules for a basic type theory

We begin by listing the rules for a basic type theory with Π, Σ, N, and Id. A point of departure is the system described by Abel et al. in [1], since a significant part of the metatheory of this system has been formalized in Agda. This system has Π-types, N and one universe. However, for better readability we use named variables instead of de Bruijn indices. We also add Σ and Id, and, in the next sections, a tower of universes.

The judgment $\Gamma \vdash$ expresses that Γ is a context. The judgment $\Gamma \vdash A$ expresses that A is a type in context Γ. The judgment $\Gamma \vdash a : A$ expresses that A is a type and a is a term of type A in context Γ. The rules are given in Figure 1.

$$\frac{}{() \vdash} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash}\ (x \text{ fresh}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash x : A}\ (x{:}A \text{ in } \Gamma)$$

Figure 1 Rules for context formation and assumption.

We may also write A type (Γ) for $\Gamma \vdash A$, and may omit the global context Γ, or the part of the context that is the same for all hypotheses and for the conclusion of the rule. Hypotheses that could be obtained from other hypotheses through inversion lemmas are often left out, for example, the hypothesis A type in the first rule for Π and Σ in Figure 2.

$$\frac{B\ \mathsf{type}\ (x : A)}{\Pi_{x:A}B\ \mathsf{type}} \qquad \frac{b : B\ (x : A)}{\lambda_{x:A}b : \Pi_{x:A}B} \qquad \frac{c : \Pi_{x:A}B \qquad a : A}{c\,a : B(a/x)}$$

$$\frac{B\ \mathsf{type}\ (x : A)}{\Sigma_{x:A}B\ \mathsf{type}} \qquad \frac{a : A \qquad b : B(a/x)}{(a, b) : \Sigma_{x:A}B} \qquad \frac{c : \Sigma_{x:A}B}{c.1 : A} \qquad \frac{c : \Sigma_{x:A}B}{c.2 : B(c.1/x)}$$

Figure 2 Rules for Π and Σ.

We write $=$ for definitional equality (or conversion). The following rules express that conversion is an equivalence relation and that judgments are invariant under conversion. The rules are given in Figures 3 and 4.

$$\frac{a : A \qquad A \ = \ B}{a : B} \qquad \frac{a \ = \ a' : A \qquad A \ = \ B}{a \ = \ a' : B}$$

$$\frac{A \ = \ B \qquad A \ = \ C}{B \ = \ C} \qquad \frac{A \ \mathsf{type}}{A \ = \ A} \qquad \frac{a \ = \ b : A \qquad a \ = \ c : A}{b \ = \ c : A} \qquad \frac{a : A}{a \ = \ a : A}$$

Figure 3 General rules for conversion.

$$\frac{A \ = \ A' \qquad B \ = \ B' \ (x : A)}{\Pi_{x:A}B \ = \ \Pi_{x:A'}B'} \qquad \frac{c \ = \ c' : \Pi_{x:A}B \qquad a \ = \ a' : A}{c\ a \ = \ c'\ a' : B(a/x)}$$

$$\frac{b : B \ (x : A) \qquad a : A}{\lambda_{x:A}b\ a \ = \ b(a/x) : B(a/x)} \qquad \frac{f\ x = g\ x : B\ (x : A)}{f = g : \Pi_{x:A}B}$$

$$\frac{A \ = \ A' \qquad B \ = \ B' \ (x : A)}{\Sigma_{x:A}B \ = \ \Sigma_{x:A'}B'} \qquad \frac{c \ = \ c' : \Sigma_{x:A}B}{c.1 \ = \ c'.1 : A} \qquad \frac{c \ = \ c' : \Sigma_{x:A}B}{c.2 \ = \ c'.2 : B(c.1/x)}$$

$$\frac{a : A \qquad b : B(a/x)}{(a,b).1 \ = \ a : A} \qquad \frac{a : A \qquad b : B(a/x)}{(a,b).2 \ = \ b : B(a/x)} \qquad \frac{c.1 \ = \ c'.1 : A \qquad c.2 \ = c'.2 : B(c.1/x)}{c \ = \ c' : \Sigma_{x:A}B}$$

Figure 4 Conversion rules for Π and Σ.

By now we have introduced several parametrized syntactic constructs for types and terms, such as $\Pi_{x:A}B$, $\lambda_{x:A}b$, $c\,a$, $(a,b).2$. Conversion rules for Π and Σ were given in Figure 4. and those rules imply that $=$ is a congruence.(Some cases of congruence are subtle. Exercise: show congruence of $=$ for $\lambda_{x:A}b$ and (a,b).) In the sequel we will tacitly assume the inference rules ensuring that $=$ is a congruence for all syntactic constructs that are to follow.

We now introduce the type of natural numbers N with the usual constructors $\mathsf{0}, \mathsf{S}$ and eliminator R, as an example of an inductive data type. Rules with the same hypotheses are written as one rule with several conclusions. The rules are given in Figure 5.

We also add identity types $\mathsf{Id}(A, a, a')$ for all A type, $a : A$ and $a' : A$, with constructor $\mathsf{refl}(A, a)$ and (based) eliminator $\mathsf{J}(A, a, C, d, a', q)$. The rules are given in Figure 6.

In this basic type theory we can define, for example, $\mathsf{isContr}(A) := \Sigma_{a:A}\Pi_{x:A}\mathsf{Id}(A, a, x)$ for A type, expressing that A is contractible. If also B type, we can define $\mathsf{Equiv}(A, B) := \Sigma_{f:A\to B}\Pi_{b:B}\mathsf{isContr}(\Sigma_{x:A}\mathsf{Id}(B, b, f(x)))$, which is the type of equivalences from A to B. This example will also be used later on.

$$\frac{}{\mathsf{N}\ \mathsf{type} \qquad \mathsf{0} : \mathsf{N}} \qquad \frac{n : \mathsf{N}}{\mathsf{S}(n) : \mathsf{N}} \qquad \frac{P\ \mathsf{type}\ (x : \mathsf{N}) \qquad a : P(\mathsf{0}/x) \qquad g : \Pi_{x:\mathsf{N}}(P \to P(\mathsf{S}(x)/x))}{\mathsf{R}(P, a, g, \mathsf{0}) = a : P(\mathsf{0}/x)}$$

$$\frac{P\ \mathsf{type}\ (x : \mathsf{N}) \qquad a : P(\mathsf{0}/x) \qquad g : \Pi_{x:\mathsf{N}}(P \to P(\mathsf{S}(x)/x)) \qquad n : \mathsf{N}}{\mathsf{R}(P, a, g, n) : P(n/x) \qquad \mathsf{R}(P, a, g, \mathsf{S}(n)) = g\ n\ \mathsf{R}(P, a, g, n) : P(\mathsf{S}(n)/x)}$$

Figure 5 Rules and conversion rules for the datatype N.

$$\frac{A\ \mathsf{type}\quad a : A\quad a' : A}{\mathsf{Id}(A,a,a')\ \mathsf{type}}\qquad\qquad \frac{a : A}{\mathsf{refl}(A,a) : \mathsf{Id}(A,a,a)}$$

$$\frac{a : A\quad C\ \mathsf{type}\ (x : A, p : \mathsf{Id}(A,a,x))\quad d : C(a/x,\mathsf{refl}(A,a)/p)\quad a' : A\quad q : \mathsf{Id}(A,a,a')}{\mathsf{J}(A,a,C,d,a',q) : C(a'/x,q/p)\qquad \mathsf{J}(A,a,C,d,a,\mathsf{refl}(A,a)) = d : C(a/x,\mathsf{refl}(A,a)/p)}$$

Figure 6 Rules and conversion rule for identity types.

3 Rules for an external sequence of universes

We present an external sequence of universes of codes of types, together with the decoding functions. (We do not include rules for cumulativity here, leaving them for Appendix A.) The rules are given in Figure 7.

$$\frac{}{\mathsf{U}_m\ \mathsf{type}}\qquad \frac{A : \mathsf{U}_m}{\mathsf{T}_m(A)\ \mathsf{type}}\qquad \frac{}{\mathsf{U}^n_m : \mathsf{U}_n\qquad \mathsf{T}_n(\mathsf{U}^n_m) = \mathsf{U}_m}\ (n > m)$$

Figure 7 Rules and conversion rules for all universes U_m and their codes U^n_m $(n > m)$.

Here and below m and n, as super- and subscripts of U and T, are *external* natural numbers, and $n \vee m$ is the maximum of n and m. This means, for example, that U_m type is a *schema*, yielding one rule for each m.

Next we define how Π, Σ, N, and Id are "relativized" to codes of types, and how they are decoded, in Figures 8 and 9.

$$\frac{A : \mathsf{U}_n\qquad B : \mathsf{T}_n(A) \to \mathsf{U}_m}{\Pi^{n,m} A B : \mathsf{U}_{n\vee m}\qquad \mathsf{T}_{n\vee m}\ (\Pi^{n,m} A B) = \Pi_{x:\mathsf{T}_n(A)} \mathsf{T}_m(B\ x)}$$

$$\frac{A : \mathsf{U}_n\qquad B : \mathsf{T}_n(A) \to \mathsf{U}_m}{\Sigma^{n,m} A B : \mathsf{U}_{n\vee m}\qquad \mathsf{T}_{n\vee m}\ (\Sigma^{n,m} A B) = \Sigma_{x:\mathsf{T}_n(A)} \mathsf{T}_m(B\ x)}$$

Figure 8 Rules and conversion rules for Π and Σ for codes of types.

$$\frac{}{\mathsf{N}^n : \mathsf{U}_n}\qquad \frac{}{\mathsf{T}_n(\mathsf{N}^n) = \mathsf{N}}$$

$$\frac{A : \mathsf{U}_n\quad a_0 : \mathsf{T}_n(A)\quad a_1 : \mathsf{T}_n(A)}{\mathsf{Id}^n(A,a_0,a_1) : \mathsf{U}_n\qquad \mathsf{T}_n(\mathsf{Id}^n(A,a_0,a_1)) = \mathsf{Id}(\mathsf{T}_n(A),a_0,a_1)}$$

Figure 9 Rules and conversion rules for codes of N and Id.

In the following section we present a type theory with *internal* universe level expressions. This theory has finitely many inference rules.

4 A type theory with universe levels and polymorphism

The problem with the type system with an external sequence of universes is that we have to *duplicate* definitions that follow the same pattern. For instance, we have the identity function

$$\mathsf{id}_n := \lambda_{X:\mathsf{U}_n}\lambda_{x:\mathsf{T}_n(X)}x : \Pi_{X:\mathsf{U}_n}\mathsf{T}_n(X) \to \mathsf{T}_n(X)$$

This is a schema that may have to be defined (and type-checked) for several n. We address this issue by introducing *universe level* expressions: we write $\alpha, \beta, \ldots$ for *level variables*, and $l, m, \ldots$ for *level expressions* which are built from level variables by suprema $l \vee m$ and the next level operation l^+. Level expressions form a sup-semilattice $l \vee m$ with a next level operation l^+ such that $l \vee l^+ = l^+$ and $(l \vee m)^+ = l^+ \vee m^+$. (We don't need a 0 element.) We write $l \leqslant m$ for $l \vee m = m$ and $l < m$ for $l^+ \leqslant m$. See [4] for more details.

We have a new context extension operation that adds a fresh level variable α to a context, a rule for assumption, and typing rules for level expressions, in Figure 10.

$$\frac{\Gamma \vdash}{\Gamma, \alpha\ \mathsf{level} \vdash}\ (\alpha \text{ fresh}) \qquad \frac{\Gamma \vdash}{\Gamma \vdash \alpha\ \mathsf{level}}(\alpha \text{ in } \Gamma) \qquad \frac{l\ \mathsf{level} \qquad m\ \mathsf{level}}{l \vee m\ \mathsf{level}} \qquad \frac{l\ \mathsf{level}}{l^+\ \mathsf{level}}$$

Figure 10 Rules for typing level expressions, extending Figure 1.

We also have level equality judgments $\Gamma \vdash l = m$ and want to enforce that judgments are invariant under level equality. To this end we add the rule that $\Gamma \vdash l = m$ when $\Gamma \vdash l$ level and $\Gamma \vdash m$ level and $l = m$ in the free sup-semilattice above with $\_^+$ and generators (level variables) in Γ.

In the next section we will also consider *hypothetical* level equality judgments, i.e., we may have constraints in Γ, quotienting the free sup-semilattice above.

We tacitly assume additional rules ensuring that level equality implies definitional equality of types and terms. It then follows from the rules of our basic type theory that judgments are invariant under level equality: if $l = m$ and $a(l/\alpha) : A(l/\alpha)$, then $a(m/\alpha) : A(m/\alpha)$.

We will now add rules for internally indexed universes in Figure 11. Note that $l < m$ is shorthand for the level equality judgment $m = l^+ \vee m$.

$$\frac{l\ \mathsf{level}}{\mathsf{U}_l\ \mathsf{type}} \qquad \frac{A : \mathsf{U}_l}{\mathsf{T}_l(A)\ \mathsf{type}} \qquad \frac{l < m}{\mathsf{U}_l^m : \mathsf{U}_m \qquad \mathsf{T}_m(\mathsf{U}_l^m) = \mathsf{U}_l}$$

Figure 11 Rules and conversion rule for universes U_l and their codes.

The remaining rules are completely analogous to the rules in Figure 8 and Figure 9 for externally indexed universes with external numbers replaced by internal levels. (To rules without assumptions, such as the first two in Fig. 9, we need to add assumptions like n level, for other rules these assumptions can be obtained from inversion lemmas.)

We expect that normalisation holds for this system. This would imply decidable type-checking. This would also imply that if $a : \mathsf{N}$ in a context with only level variables, then a is convertible to a numeral.

Interpreting the level-indexed system in the system with externally indexed universes

A judgment in the level-indexed system can be interpreted in the externally indexed system relative to an assignment ρ of external natural numbers to level variables. We simply replace each level expression in the judgment by the corresponding natural number obtained by letting $l^+\,\rho = l\,\rho + 1$ and $(l \vee m)\,\rho = \max(l\,\rho, m\,\rho)$.

Rules for level-indexed products

In Agda Level is a type, and it is thus possible to form level-indexed products of types as Π-types. In our system this is not possible, since level is not a type. Nevertheless, it is useful for modularity to be able to form level-indexed products. Thus we extend the system with the rules in Figure 12.

$$\frac{A\ \mathsf{type}\ (\alpha\ \mathsf{level})}{[\alpha]A\ \mathsf{type}} \qquad \frac{t : [\alpha]A \qquad l\ \mathsf{level}}{t\ l : A(l/\alpha)} \qquad \frac{u : A\ (\alpha\ \mathsf{level})}{\langle\alpha\rangle u : [\alpha]A} \qquad \frac{t\ \alpha = u\ \alpha : A\ (\alpha\ \mathsf{level})}{t = u : [\alpha]A}$$

$$\frac{u : A\ (\alpha\ \mathsf{level}) \qquad l\ \mathsf{level}}{(\langle\alpha\rangle u)\ l = u(l/\alpha) : A(l/\alpha)}$$

Figure 12 Rules and conversion rule for level-indexed products.

In this type theory we can reflect, for example, $\mathsf{isContr}(A) := \Sigma_{a:A}\Pi_{x:A}\mathsf{Id}(A,a,x)$ for A type as follows. In the context $\alpha\,\mathsf{level}, A : \mathsf{U}_\alpha$, define

$$\mathsf{isContr}^\alpha(A) := \Sigma^{\alpha,\alpha}\ A\,(\lambda_{a:\mathsf{T}_\alpha(A)}(\Pi^{\alpha,\alpha}\ A\,(\lambda_{x:\mathsf{T}_\alpha(A)}\mathsf{Id}^\alpha(A,a,x)))).$$

Then $\mathsf{T}_\alpha(\mathsf{isContr}^\alpha(A)) = \mathsf{isContr}(\mathsf{T}_\alpha(A))$. We can further abstract to obtain the following typing:

$$\langle\alpha\rangle\lambda_{A:\mathsf{U}_\alpha}\mathsf{isContr}^\alpha(A) : [\alpha](\mathsf{U}_\alpha \to \mathsf{U}_\alpha).$$

In a similar way we can reflect $\mathsf{Equiv}(A,B)$ for $A, B\,\mathsf{type}$ by defining in context $\alpha\,\mathsf{level}, \beta\,\mathsf{level},\ A : \mathsf{U}_\alpha, B : \mathsf{U}_\beta$ a term $\mathsf{Eq}^{\alpha,\beta}(A,B) : \mathsf{U}_{\alpha\vee\beta}$ such that $\mathsf{T}_{\alpha\vee\beta}(\mathsf{Eq}^{\alpha,\beta}(A,B)) = \mathsf{Equiv}(\mathsf{T}_\alpha(A), \mathsf{T}_\beta(B))$.

An example that uses level-indexed products beyond the ML-style polymorphism (provided by Sozeau and Tabareau and by Voevodsky) is the following type which expresses the theorem that univalence for universes of arbitrary level implies function extensionality for functions between universes of arbitrary levels.

$$([\alpha]\mathsf{IsUnivalent}\,\mathsf{U}_\alpha) \to [\beta][\gamma]\mathsf{FunExt}\,\mathsf{U}_\beta\,\mathsf{U}_\gamma$$

In other words, *global* univalence implies *global* function extensionality.

Since an assumption of global function extensionality can replace many assumptions of local function extensionality (provided by ML-style polymorphism), this can also give rise to shorter code, see the example `Eq-Eq-cong'` in [13].

5 A type theory with level constraints

To motivate why it may be useful to introduce the notion of judgment relative to a list of constraints on universe levels, consider the following type in a system without cumulativity. (We use Russell style notation for readability, see Appendix B for the rules for the Russell style version of our system.)

$$\Pi_{A:\mathsf{U}_l\ B:\mathsf{U}_m\ C:\mathsf{U}_n}\ \ \mathsf{Id}\,\mathsf{U}_{l\vee m}\,(A\times B)\,(C\times A)\to\mathsf{Id}\,\mathsf{U}_{m\vee l}\,(B\times A)\,(C\times A)$$

This is well-formed provided $l \vee m = n \vee l$. There are several independent solutions:

$$l = \alpha, m = \beta, n = \alpha \vee \beta$$
$$l = \alpha, m = \gamma \vee \alpha, n = \gamma$$
$$l = \beta \vee \gamma, m = \beta, n = \gamma$$
$$l = \alpha, m = \beta, n = \beta$$

where α, β, and γ are level variables. It should be clear that there cannot be any *most general* solution, since this solution would have to assign variables to l, m, n.

In a system with level constraints, we could instead derive the (inhabited under UA) type

$$\Pi_{A:\mathsf{U}_\alpha\ B:\mathsf{U}_\beta\ C:\mathsf{U}_\gamma}\ \ \mathsf{Id}\,\mathsf{U}_{\alpha\vee\beta}\,(A\times B)\,(C\times A)\to\mathsf{Id}\,\mathsf{U}_{\alpha\vee\gamma}\,(B\times A)\,(C\times A)$$

which is valid under the constraint $\alpha \vee \beta = \alpha \vee \gamma$, which captures all solutions simultaneously.

Without being able to declare explicitely such constraints, one would instead need to write four separate definitions.

Surprisingly, if we add a least level 0 to the term levels (like in Agda) then there *is* a most general solution, namely $l = \alpha \vee \beta \vee \delta,\ m = \beta \vee \gamma,\ n = \alpha \vee \gamma$,[1] since it can be seen as an instance of a Associative Commutative Unit Idempotent unification problem [3].

It is however possible to find equation systems which do not have a most general unifier, even with a least level 0, using the next level operation. For instance, the system $l^+ = m \vee n$ does not have a most general unifier, using a reasoning similar to the one in [15].

Rules for level constraints

A constraint is an equation $l = m$, where l and m are level expressions. Voevodsky [39] suggested to introduce universe levels with constraints. This corresponds to mathematical practice: for instance, at the beginning of the book [17], the author introduces two universes U and V with the constraint that U is a member of V. In our setting this will correspond to introducing two levels α and β with the constraint $\alpha < \beta$.

Note that $\alpha < \beta$ holds iff $\beta = \beta \vee \alpha^+$. We can thus avoid declaring this constraint if we instead systematically replace β by $\beta \vee \alpha^+$. This is what is currently done in the system Agda. However, this is a rather indirect way to express what is going on. Furthermore, the example at the beginning of this section shows that this can lead to an artificial duplication of definitions.

Recall that we have in Section 4, e.g., the rule that $\mathsf{U}_l^m : \mathsf{U}_m$ if $l < m$ valid, that is, if $l < m$ holds in the free semilattice. In the extended system in this section, this typing rule also applies when $l < m$ is implied by the constraints in the context Γ. For instance, we have $\alpha^+ \leqslant \beta$ in a context with constraints $\alpha \leqslant \gamma$ and $\gamma^+ \leqslant \beta$.

To this end we introduce a new context extension operation Γ, ψ extending a context Γ by a finite set of constraints ψ. The first condition for forming Γ, ψ is that all level variables occurring in ψ are declared in Γ. The second condition is that the finite set of constaints in the extended context Γ, ψ is loop-free. A finite set of constraints is *loop-free* if it does not create a *loop*, i.e., a level expression l such that $l < l$ modulo this set of constraints, see [4].

[1] We learnt this from Thiago Felicissimo, with a reference to the work [15].

We also have a new judgment form $\Gamma \vdash \psi \ \mathsf{valid}$ that expresses that the constraints in ψ hold in Γ, that is, are implied by the constraints in Γ. If there are no constraints in Γ, the judgment $\Gamma \vdash \{l = m\} \ \mathsf{valid}$ amounts to the same as $\Gamma \vdash l = m$ in Section 4. Otherwise it means that the constraints in ψ hold in the sup-semilattice with $\_^+$ presented by Γ.

As shown in [4], $\Gamma \vdash \psi \ \mathsf{valid}$ as well as loop-checking, is decidable in polynomial time.

Voevodsky [39] did not describe a mechanism to *eliminate* universe levels and constraints. In Figure 12 we gave rules for eliminating universe levels and in Figure 13 below we give rules for eliminating universe level constraints.

Rules for constraint-indexed products

We introduce a "restriction" or "constraining" operation with the rules in Figure 13.

$$\frac{A \ \mathsf{type} \quad (\psi)}{[\psi]A \ \mathsf{type}} \qquad \frac{t : A \quad (\psi)}{\langle\psi\rangle t : [\psi]A} \qquad \frac{\psi \ \mathsf{valid}}{[\psi]A = A} \qquad \frac{\psi \ \mathsf{valid}}{\langle\psi\rangle t = t}$$

Figure 13 Rules for constraining.

Here is a simple example of the use of this system. In order to represent set theory in type theory, we can use a type V satisfying the following equality $\mathsf{Id} \ \mathsf{U}_\beta \ V \ (\Sigma_{X:\mathsf{U}_\alpha} X \to V)$. This equation is only well-typed modulo the constraint $\alpha < \beta$.

We can define in our system a constant

$$c \ = \ \langle\alpha \ \beta\rangle\langle\alpha < \beta\rangle\lambda_{Y:\mathsf{U}_\beta}\mathsf{Id} \ \mathsf{U}_\beta \ Y \ (\Sigma_{X:\mathsf{U}_\alpha} X \to Y) \quad : \quad [\alpha \ \beta][\alpha < \beta](\mathsf{U}_\beta \to \mathsf{U}_{\beta+})$$

This is because $\Sigma_{X:\mathsf{U}_\alpha} X \to Y$ has type U_β in the context

$$\alpha : \mathsf{level}, \ \beta : \mathsf{level}, \ \alpha < \beta, \ Y : \mathsf{U}_\beta$$

We can further instantiate this constant c on two levels l and m, and this will be of type

$$[l < m](\mathsf{U}_m \to \mathsf{U}_{m^+})$$

and this can only be used further if $l < m$ holds in the current context[2].

In the current system of Agda, the constraint $\alpha < \beta$ is represented indirectly by writing β on the form $\gamma \vee \alpha^+$ and c is defined as

$$c \ = \ \langle\alpha \ \gamma\rangle\lambda_{Y:\mathsf{U}_{\alpha^+\vee\gamma}}\mathsf{Id} \ \mathsf{U}_{\alpha^+\vee\gamma} \ Y \ (\Sigma_{X:\mathsf{U}_\alpha} X \to Y) \quad : \quad [\alpha \ \gamma](\mathsf{U}_{\alpha^+\vee\gamma} \to \mathsf{U}_{\alpha^{++}\vee\gamma^+})$$

which arguably is less readable.

[2] It is interesting to replace $\mathsf{Id} \ \mathsf{U}_\beta$ in the definition of c above by Eq. We leave it to the reader to verify the following typing, for which no constraint is needed:

$$c' \ = \ \langle\alpha \ \beta\rangle\lambda_{Y:\mathsf{U}_\beta} \ \mathsf{Eq} \ Y \ (\Sigma_{X:\mathsf{U}_\alpha} X \to Y) \quad : \quad [\alpha \ \beta](\mathsf{U}_\beta \to \mathsf{U}_{\beta\vee\alpha^+})$$

In general, if we build a term t of type A in a context using labels $\alpha_1, \ldots, \alpha_m$ and constraint ψ and variables $x_1 : A_1, \ldots, x_n : A_n$ we can introduce a constant

$$c \;=\; \langle \alpha_1 \;\ldots\; \alpha_m \rangle \langle \psi \rangle \lambda_{x_1 \;\ldots\; x_n} t \;:\; [\alpha_1 \;\ldots\; \alpha_m][\psi] \Pi_{x_1 : A_1 \;\ldots\; x_n : A_n} A$$

We can then instantiate this constant $c\ l_1 \;\ldots\; l_m\ u_1 \;\ldots\; u_n$, but only if the levels $l_1 \;\ldots\; l_m$ satisfy the constraint ψ.

We remark that Voevodsky's system [39] has no constraint-indexed products and no associated application operation, and instantiation of levels is only a meta-level operation. Sozeau and Tabareau [30] do not have constraint-index products either. However, they do have a special operation for instantiating universe-polymorphic constants defined in the global environment.

▶ Remark 1. Let's discuss some special cases and variations.

First, it is possible not to use level variables at all, making the semilattice empty, in which case the type theory defaults to one without universes as presented in Section 2.

Second, one could have exactly one level variable in the context. Then any constraint would either be a loop or trivial. In the latter case, the finitely presented semilattice is isomorphic to the natural numbers with successor and max. Still, we get some more expressivity than the type theory in Section 3 since we can express universe polymorphism in one variable.

Third, with arbitrarily many level variables but not using constraints we get the type theory in Section 4.

Fourth, we could add a bottom element, or empty supremum, to the semilattice. Without level variables and constraints, the finitely presented semilattice is isomorphic to the natural numbers with successor and max and we would get the type theory in Section 3. We would also get a first universe. (Alternatively, one could have one designated level variable 0 and constraints $0 \leqslant \alpha$ for all level variables α.)

Fifth, we note in passing that the one-point semilattice with $\_^+$ has a loop.

6 Related work

We have already discussed both Coq's and Agda's treatment of universe polymorphism in the introduction, including the work of Huet, Harper and Pollack, Courant, Herbelin, and Sozeau and Tabareau, as well as of Voevodsky. In this section we further discuss the latter two, as well as some recent related work.

Lean

One can roughly describe the type system of Lean [12, 7] as our current type system where we only can declare constants of the form $c \;=\; \langle \alpha_1 \;\ldots\; \alpha_n \rangle M \;\;:\;\; [\alpha_1 \;\ldots\; \alpha_n] A$ where there are no new level variables introduced in M and A.

Voevodsky

One of our starting points was the 79 pp. draft [39] by Voevodsky, where type theories are parametrized by a fixed but arbitrary finite set of constraints over a given finite set Fu of *u-level variables*. A *u-level expression* [39, Def. 2.0.2] is either a numeral, or a variable in Fu, or an expression of the form $M + n$ with n a numeral and M a u-level expression, or of the form $\max(M_1, M_2)$ with M_1, M_2 u-level expression. A *constraint* is an equation between two u-level expressions. Given the finite set of constraints, $\mathcal{A}$ is the set of assigments of natural numbers to variables in Fu that satisfy all constraints.

The rules 7 and 10 in [39, Section 3.4] define how to use constraints: two types (and, similarly, two terms) become definitionally equal if, for all assignments in $\mathcal{A}$, the two types become *essentially* syntactically equal after substitution of all variables in *Fu* by their assigned natural number. For example, the constraint $\alpha < \beta$ makes U_β and $\mathsf{U}_{\max(1,\beta)}$ definitionally equal.

For decidability, Voevodsky refers in the proof of [39, Lemma 2.0.4, proof] to Presburger Arithmetic, in which his constraints can easily be expressed.[3]This indeed implies that definitional equality is decidable, even "in practice [...] expected to be very easily decidable i.e. to have low complexity of the decision procedure" [39, p. 5, l. -13]. The latter is confirmed by [4].

The remaining sections of [39] are devoted to extending the type theory with data types, W-types and identity types, and to its metatheory.

We summarize the main differences between our type theories and Voevodsky's as follows. In [39], u-levels are natural numbers, even though u-level expressions can also contain u-level variables, successor and maximum. Our levels are elements of an abstract sup-semilattice with a successor operation. In the abstract setting, for example, $\alpha \vee \beta = \alpha^+$ does not imply $\beta = \alpha^+$, whereas in [39] it does. In [39], constraints are introduced, once and for all, at the level of the theory. In our proposal they are introduced at the level of contexts. There are no level-indexed products and no constraint-indexed products in [39]. We also remark that Voevodsky's system is Tarski-style and has cumulativity (rules 29 and 30 in [39, Section 3.4]). Our system is also Tarski-style, but we present a Russell-style version in Appendix B. We present rules for cumulativity in Appendix A.

Sozeau and Tabareau

In Sozeau and Tabareau's [30] work on universe polymorphism in the Coq tradition, there are special rules for introducing universe-monomorphic and universe-polymorphic constants, as well as a rule for instantiating the latter. However, their system does not include the full explicit universe polymorphism provided by level- and constraint-indexed products. In our system, with explicit universe polymorphism, we can have a uniform treatment of definitions, all of the form

$$c : A = t$$

where A is a type and t a term of type A, and these definitions can be local as well.

The constraint languages differ: their constraints are equalities or (strict) inequalities between level variables, while ours are equalities between level expressions generated by the supremum and successor operations.

Furthermore, they consider cumulative universe hierarchies à la Russell, while our universes are à la Tarski and we consider both non-cumulative (like Agda) and cumulative versions.

One further important difference is that their system has been completely implemented and tested on significant examples, while our system is at this stage only a proposal. The idea would be that the users have to declare explicitly both universe levels *and* constraints. The Agda implementation shows that it works in practice to be explicit about universe levels, and we expect that to be explicit about constraints will actually simplify the use of the system, but this has yet to be tested in practice. Recently, Coq has been extended to support universes and constraint annotations from entirely implicit to explicit. Moreover, our level- and constraint-indexed products can to some extent be simulated by using Coq's module system [29].

[3] For this it seems necessary to also require that $\mathcal{A}$ is defined by a *finite* set of constraints.

Assaf and Thiré

Assaf [2] considers an alternative version of the calculus of constructions where subtyping is explicit. This new system avoids problems related to coercions and dependent types by using the Tarski style of universes and by introducing additional equations to reflect equality. In particular he adds an explicit cumulativity map $\mathsf{T}_1^0 : \mathsf{U}_0 \to \mathsf{U}_1$. He argues that "full reflection" is necessary to achieve the expressivity of Russell style. He introduces the explicit cumulative calculus of constructions (CC↑) which is closely related to our system of externally indexed Tarski style universes. This is analysed further in the PhD thesis of F. Thiré [38].

7 Conjectures and future work

Canonicity and normalization have been proved for a type theory with an external tower of universes [10]. We conjecture that these proofs can be modified to yield proofs of analogous properties (and their corollaries) for our type theories in Section 4 and 5. In particular, decidability of type checking should follow using [4].

References

1 Andreas Abel, Joakim Öhman, and Andrea Vezzosi. Decidability of conversion for type theory in type theory. *Proceedings of the ACM on Programming Languages*, 2(POPL):23:1–23:29, 2018. `doi:10.1145/3158111`.

2 Ali Assaf. A calculus of constructions with explicit subtyping. In *20th International Conference on Types for Proofs and Programs, TYPES*, pages 27–46, 2014.

3 Franz Baader and Jörg H. Siekmann. Unification theory. In *Handbook of Logic in Artificial Intelligence and Logic Programming, Volume 2, Deduction Methodologies*, pages 41–126. Oxford University Press, 1994.

4 Marc Bezem and Thierry Coquand. Loop-checking and the uniform word problem for join-semilattices with an inflationary endomorphism. *Theoretical Computer Science*, 913:1–7, 2022. `doi:10.1016/j.tcs.2022.01.017`.

5 Marc Bezem, Thierry Coquand, Peter Dybjer, and Martín Escardó. The Burali-Forti argument in HoTT/UF with applications to the type of groups in a universe. `https://www.cs.bham.ac.uk/~mhe/TypeTopology/Ordinals.BuraliForti.html`, 2022.

6 Luca Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8(2):147–172, 1987. `doi:10.1016/0167-6423(87)90019-0`.

7 Mario Carneiro. The type theory of Lean. Master Thesis, Carnegie-Mellon University, 2019.

8 Antoine Chambert-Loir. A presheaf that has no associated sheaf. `https://freedommathdance.blogspot.com/2013/03/a-presheaf-that-has-no-associated-sheaf.html`, 2013.

9 Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

10 Thierry Coquand. Canonicity and normalization for dependent type theory. *Theoretical Computer Science*, 777:184–191, 2019. `doi:10.1016/j.tcs.2019.01.015`.

11 Judicaël Courant. Explicit universes for the calculus of constructions. In *Theorem Proving in Higher Order Logics, TPHOLs*, volume 2410 of *Lecture Notes in Computer Science*, pages 115–130. Springer, 2002. `doi:10.1007/3-540-45685-6_9`.

12 Leonardo de Moura, Soonho Kong, Jeremy Avigad, Floris van Doorn, and Jakob von Raumer. The Lean theorem prover (system description). In *Conference on Automated Deduction (CADE-25)*, volume 9195 of *Lecture Notes in Computer Science*, pages 378–388, 2015.

13 Martín Hötzel Escardó. Introduction to univalent foundations of mathematics with Agda. *CoRR*, 2019. `arXiv:1911.00580`.

14 Martín Hötzel Escardó et al. TypeTopology. `https://www.cs.bham.ac.uk/~mhe/TypeTopology/index.html`. Agda development.

15 Thiago Felicissimo, Frédéric Blanqui, and Ashish Kumar Barnawal. Translating proofs from an impredicative type system to a predicative one. In *Computer Science Logic (CSL)*, 2023.

16 Jean-Yves Girard. *Thèse d'État*. PhD thesis, Université Paris VII, 1971.

17 Jean Giraud. *Cohomologie non abélienne*. Springer, 1971. doi:10.1007/978-3-662-62103-5.

18 Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89:107–136, 1991.

19 Hugo Herbelin. Type inference with algebraic universes in the Calculus of Inductive Constructions. http://pauillac.inria.fr/~herbelin/articles/univalgcci.pdf, 2005.

20 Gérard Huet. Extending the calculus of constructions with Type:Type. unpublished manuscript, April 1987.

21 Per Martin-Löf. On the strength of intuitionistic reasoning. Preprint, Stockholm University, 1971.

22 Per Martin-Löf. A theory of types. Preprint, Stockholm University, 1971.

23 Per Martin-Löf. An intuitionistic theory of types. Preprint, Stockholm University, 1972.

24 Per Martin-Löf. An intuitionistic theory of types: Predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, pages 73–118. North Holland, 1975.

25 Per Martin-Löf. Constructive mathematics and computer programming. In *Logic, Methodology and Philosophy of Science, VI, 1979*, pages 153–175. North-Holland, 1982.

26 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

27 Egbert Rijke, Elisabeth Bonnevier, Jonathan Prieto-Cubides, Fredrik Bakke, and others. Univalent mathematics in Agda. https://github.com/UniMath/agda-unimath/.

28 Carlos Simpson. Computer theorem proving in mathematics. *Letters in Mathematical Physics*, 69(1-3):287–315, July 2004. doi:10.1007/s11005-004-0607-9.

29 Matthieu Sozeau. Explicit universes. https://coq.inria.fr/refman/addendum/universe-polymorphism.html#explicit-universes.

30 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in Coq. In *Interactive Theorem Proving (ITP)*, 2014.

31 Thomas Streicher. *Semantics of Type Theory*. Birkhäuser, 1991.

32 Agda team. The Agda manual. URL: https://agda.readthedocs.io/en/v2.6.2.1/.

33 The 1Lab Development Team. The 1Lab. https://1lab.dev.

34 The Agda Community. Agda standard library. https://github.com/agda/agda-stdlib.

35 The Coq Community. Coq. https://coq.inria.fr.

36 The Cubical Agda Community. A standard library for Cubical Agda. https://github.com/agda/cubical.

37 The HoTT-Agda Community. HoTT-Agda. https://github.com/HoTT/HoTT-Agda.

38 François Thiré. *Interoperability between proof systems using the logical framework Dedukti. (Interopérabilité entre systèmes de preuve en utilisant le cadre logique Dedukti)*. PhD thesis, École normale supérieure Paris-Saclay, Cachan, France, 2020. URL: https://tel.archives-ouvertes.fr/tel-03224039.

39 Vladimir Voevodsky. Universe polymorphic type system. http://www.math.ias.edu/Voevodsky/voevodsky-publications_abstracts.html#UPTS, 2014.

40 William C. Waterhouse. Basically bounded functors and flat sheaves. *Pacific Math. J*, 57(2):597–610, 1975.

A Formulation with cumulativity

We introduce an operation $\mathsf{T}_l^m(A) : \mathsf{U}_m$ if $A : \mathsf{U}_l$ and $l \leqslant m$ (i.e., $m = l \vee m$).[4]

We require $\mathsf{T}_m(\mathsf{T}_l^m(A)) = \mathsf{T}_l(A)$. Note that this yields, e.g., $a : \mathsf{T}_m(\mathsf{T}_l^m(A))$ if $a : \mathsf{T}_l(A)$. We also require $\mathsf{T}_l^m(\mathsf{N}^l) = \mathsf{N}^m$ $(l \leqslant m)$, and $\mathsf{T}_l^m(\mathsf{U}_k^l) = \mathsf{U}_k^m$ $(k < l \leqslant m)$, as well as $\mathsf{T}_l^m(A) = A$ $(l = m)$ and $\mathsf{T}_m^n(\mathsf{T}_l^m(A)) = \mathsf{T}_l^n(A)$ $(l \leqslant m \leqslant n)$, for all $A : \mathsf{U}_l$.

[4] Recall that the equality of universe levels is the one of sup-semilattice with the $\_^+$ operation.

We can then simplify the product and sum rules to

$$\frac{A : \mathsf{U}_l \qquad B : \mathsf{T}_l(A) \to \mathsf{U}_l}{\Pi^l AB : \mathsf{U}_l} \qquad\qquad \frac{A : \mathsf{U}_l \qquad B : \mathsf{T}_l(A) \to \mathsf{U}_l}{\Sigma^l AB : \mathsf{U}_l}$$

with conversion rules

$$\mathsf{T}_l\ (\Pi^l AB) = \Pi_{x:\mathsf{T}_l(A)}\mathsf{T}_l(B\ x) \qquad \mathsf{T}_l\ (\Sigma^l AB) = \Sigma_{x:\mathsf{T}_l(A)}\mathsf{T}_l(B\ x)$$

and

$$\mathsf{T}_l^m\ (\Pi^l AB) = \Pi^m \mathsf{T}_l^m(A)(\lambda_{x:\mathsf{T}_l(A)}\mathsf{T}_l^m(B\ x)) \quad \mathsf{T}_l^m\ (\Sigma^l AB) = \Sigma^m \mathsf{T}_l^m(A)(\lambda_{x:\mathsf{T}_l(A)}\mathsf{T}_l^m(B\ x))$$

Recall the family $\mathsf{Id}^l(A, a, b) : \mathsf{U}_l$ for $A : \mathsf{U}_l$ and $a : \mathsf{T}_l(A)$ and $b : \mathsf{T}_l(B)$, with judgemental equality $\mathsf{T}_l(\mathsf{Id}^l(A, a, b)) = \mathsf{Id}(\mathsf{T}_l(A), a, b)$. We add the judgmental equalities $\mathsf{T}_l^m(\mathsf{Id}^l(A, a, b)) = \mathsf{Id}^m(\mathsf{T}_l^m(A), a, b)$; note that a and b are well-typed since $\mathsf{T}_m(\mathsf{T}_l^m(A)) = \mathsf{T}_l(A)$.

Example. Recall the type $\mathsf{Eq}^{l,l}(A, B) : \mathsf{U}_l$ for A and B in U_l, with judgmental equality $\mathsf{T}_l(\mathsf{Eq}^{l,l}(A, B)) = \mathsf{Equiv}(\mathsf{T}_l(A), \mathsf{T}_l(B))$. For $m > l$, a consequence of univalence for U_m and U_l is that we can build an element of the type

$$\mathsf{Id}(\mathsf{U}_m, \mathsf{Eq}^{m,m}(\mathsf{T}_l^m(A), \mathsf{T}_l^m(B)), \mathsf{Id}^m(\mathsf{U}_l^m, A, B)).$$

B Notions of model and formulation à la Russell

Generalised algebraic presentation

In a forthcoming paper, we plan to present some generalised algebraic theories of level-indexed categories with families with extra structure. The models of these theories provide suitable notions of model of our type theories with level judgments. Moreover, the theories presented in this paper are initial objects in categories of such models.

▶ Remark 2. As explained in [31], in order to see the theories in this paper as presenting *initial* models, it is enough to use a variation where application $c\ a : B(a/x)$ for $c : \Pi_{x:A}B$ and $a : A$ is annoted by the type family A, B (and similarly for the pairing operation). If the theories satisy the normal form property, it can then be shown that also the theories without annotated application are initial.

Russell formulation

Above, we presented type theories with universe level judgments *à la Tarski*. There are alternative formulations *à la Russell* (using the terminology introduced in [26] of universes). One expects these formulations to be equivalent to the Tarski-versions, and thus also initial models. For preliminary results in this direction see [2, 38].

With this formulation, the version without cumulativity becomes

$$\frac{A : \mathsf{U}_n}{A\ \mathsf{type}}$$

$$\frac{A : \mathsf{U}_n \qquad B : \mathsf{U}_m(x : A)}{\Pi_{x:A}B : \mathsf{U}_{n\vee m}} \qquad\qquad \frac{A : \mathsf{U}_n \qquad B : \mathsf{U}_m(x : A)}{\Sigma_{x:A}B : \mathsf{U}_{n\vee m}}$$

$$\frac{l\ \mathsf{level}}{\mathsf{N} : \mathsf{U}_l}$$

$$\frac{A : \mathsf{U}_n \qquad a_0 : A \qquad a_1 : A}{\mathsf{Id}(A, a_0, a_1) : \mathsf{U}_n}$$

$$\frac{l < n}{\mathsf{U}_l : \mathsf{U}_n}$$

For the version with cumulativity, we add the rules

$$\frac{A : \mathsf{U}_l \qquad l \leqslant n}{A : \mathsf{U}_n}$$

and the rules for products and sums can be simplified to

$$\frac{A : \mathsf{U}_n \qquad B : \mathsf{U}_n\ (x : A)}{\Pi_{x:A} B : \mathsf{U}_n} \qquad\qquad \frac{A : \mathsf{U}_n \qquad B : \mathsf{U}_n\ (x : A)}{\Sigma_{x:A} B : \mathsf{U}_n}$$

For $m > l$ the consequence of univalence for U_m and U_l mentioned in Appendix A can now be written simply as

$$\mathsf{Id}(\mathsf{U}_m, \mathsf{Equiv}(A, B), \mathsf{Id}(\mathsf{U}_l, A, B)).$$

▶ Remark 3. In the version à la Tarski, with or without cumulativity, terms have unique types, in the sense that if $t : A$ and $t : B$ then $A = B$, by induction on t. But for this to be valid, we need to annotate application as discussed in Remark 2. Even with annotated application, the following property is not elementary: if U_n and U_m are convertible then n is equal to m. This kind of property is needed for showing the equivalence between the Tarski and the Russell formulation.

▶ Remark 4. If, in a system without cumulativity, we extend our system of levels with a least level 0, then if we restrict N to be of type U_0, and U_n to be of type U_{n+1} then well formed terms have unique types.

▶ Remark 5. It should be the case that the above formulation à la Russell presents the initial CwF with extra extructure for the standard type formers and a hierarchy of universes, but the proof doesn't seem to be trivial, due to Remark 3.

A Univalent Formalization of Constructive Affine Schemes

Max Zeuner
Department of Mathematics, Stockholm University, Sweden

Anders Mörtberg
Department of Mathematics, Stockholm University, Sweden

Abstract

We present a formalization of constructive affine schemes in the Cubical Agda proof assistant. This development is not only fully constructive and predicative, it also makes crucial use of univalence. By now schemes have been formalized in various proof assistants. However, most existing formalizations follow the inherently non-constructive approach of Hartshorne's classic "Algebraic Geometry" textbook, for which the construction of the so-called structure sheaf is rather straightforwardly formalizable and works the same with or without univalence. We follow an alternative approach that uses a point-free description of the constructive counterpart of the Zariski spectrum called the Zariski lattice and proceeds by defining the structure sheaf on formal basic opens and then lift it to the whole lattice. This general strategy is used in a plethora of textbooks, but formalizing it has proved tricky. The main result of this paper is that with the help of the univalence principle we can make this "lift from basis" strategy formal and obtain a fully formalized account of constructive affine schemes.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Constructive mathematics; Theory of computation → Type theory

Keywords and phrases Affine Schemes, Homotopy Type Theory and Univalent Foundations, Cubical Agda, Constructive Mathematics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.14

Related Version *Full Version*: https://arxiv.org/abs/2212.02902

Supplementary Material *Software (Agda Source Code)*: https://github.com/agda/cubical/blob/310a0956bb45ea49a5f0aede0e10245292ae41e0/Cubical/Papers/AffineSchemes.agda
archived at swh:1:cnt:beb6a5859db19530348074be6e9166c04871e6c7

Funding This paper is based upon research supported by the Swedish Research Council (SRC, Vetenskapsrådet) under Grant No. 2019-04545. The research has also received funding from the Knut and Alice Wallenberg Foundation through the Foundation's program for mathematics.

Acknowledgements We would like to thank Thierry Coquand for his continued feedback and invaluable comments throughout this project. We are also indebted to Felix Cherubini for his comments and his work on the Cubical Agda library, particularly for his ring solver. Furthermore, we thank Martín Hötzel Escardó, Peter Dybjer, Peter LeFanu Lumsdaine, Egbert Rijke and the participants of the "Proof and Computation" autumn school in Fischbachau for our discussions.

1 Introduction

Algebraic geometry originated as the study of solutions of polynomials. Historically, the geometric objects of interest would be for example *complex affine varieties* – subsets of $\mathbb{C}^n$ defined by systems of polynomial equations. Starting with the pioneering work of Grothendieck in the 1960s, the scope of the discipline was drastically widened, making it one of the most pervasive in modern day mathematics. At the heart of this development are *schemes* – geometric objects that generalize from algebraically closed fields, like $\mathbb{C}$, to arbitrary commutative rings.

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 14; pp. 14:1–14:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

A point $a \in \mathbb{C}$ corresponds to the maximal ideal of the polynomial ring $\mathbb{C}[x]$ consisting of polynomials p such that $p(a) = 0$, i.e. the ideal generated by $(x - a)$. By looking not only at maximal ideals, but also at prime ideals of $\mathbb{C}[x]$, we arrive at the *spectrum* of $\mathbb{C}[x]$, denoted $\mathsf{Spec}\,\mathbb{C}[x]$. As $\mathbb{C}[x]$ has a non-maximal prime ideal, the zero-ideal, $\mathsf{Spec}\,\mathbb{C}[x]$ contains an additional point to $\mathbb{C}$ and carries a very different topology. This is called the *Zariski topology* in which the open sets are generated by *basic opens* $D(p) \subseteq \mathsf{Spec}\,\mathbb{C}[x]$ where $p \in \mathbb{C}[x]$. If $p \neq 0$, $D(p)$ corresponds to the set of points a where $p(a) \neq 0$ together with the zero-ideal. The spectrum can then be equipped with a *structure sheaf* that associates to every Zariski open set U a ring of "rational functions" definable on U. For a basic open $D(p)$, this will be the ring of function ${}^{q(x)}\!/_{p(x)^n}$ where q is another polynomial. This corresponds to the functions of the quotient ring $\mathbb{C}(x)$ that are definable everywhere but at the zeros of p. See Vakil's "The Rising Sea" [34, Ex. 3.2.3.1] for a more in-depth discussion of this motivating example and an illustration of $\mathsf{Spec}\,\mathbb{C}[x]$.

This construction can be carried out for any commutative ring R instead of $\mathbb{C}[x]$: the spectrum $\mathsf{Spec}\,R$ is the set of prime ideals of R and its Zariski topology is again generated by basic opens. For $f \in R$, the basic open $D(f)$ is the set of prime ideals that do *not* contain f. The structure sheaf maps $D(f)$ to the *localization* $R[{}^1\!/_f]$, the ring of fractions ${}^r\!/_{f^n}$ where $r \in R$ and the denominator is a power of f. One can prove that this always defines a sheaf, i.e. is compatible with taking covers of open sets in a certain sense.

When Grothendieck introduced the general notion of (affine) schemes, he did so in a structural fashion that is typical for his work. Mathematical objects, in particular algebraic structures, are taken to be identical if they are isomorphic in some unique, or at least canonical, way. When constructing the structure sheaf, however, this leads to a problem of well-definedness: if $D(f) = D(g)$, then we better have $R[{}^1\!/_f] = R[{}^1\!/_g]$. Unfortunately, it is not difficult to come up with examples violating this. For example, we have $D(x) = D(x^2)$ in $\mathbb{C}[x]$ (both functions vanish only at 0), but formally speaking $\mathbb{C}[x][{}^1\!/_x]$ is not strictly the same ring as $\mathbb{C}[x][{}^1\!/_{x^2}]$ despite them clearly being isomorphic and describing the same sub-ring of the quotient ring $\mathbb{C}(x)$, as ${}^1\!/_x = {}^x\!/_{x^2}$.

In this paper we show how this problem can be solved with the help of univalence. In particular, we present a formalization in `Cubical Agda` [35] of constructive affine schemes following Coquand, Lombardi and Schuster [10]. In the constructive setting, the Zariski spectrum of a commutative ring is replaced by the so-called *Zariski lattice*. Elements of this lattice are *finitely* generated by formal basic opens, which allows for a completely predicative approach that does not require additional assumptions like Voevodsky's resizing axioms [38].

The definition of constructive affine schemes still works analogously to the classical definition given in most textbooks ranging from Grothendieck's authoritative classic "EGA I" [13], to more modern treatments such as "Algebraic Geometry" by Görtz and Wedhorn [15], "The Rising Sea" by Vakil [34], or Johnstone's "Stone Spaces" [18]. In either case one starts with the basic opens, on which the structure sheaf is defined and proved to be a sheaf. Using abstract categorical machinery this is then lifted to a sheaf on the whole Zariski spectrum/lattice. More precisely, one takes the right Kan extension along the inclusion of basic opens, which preserves the sheaf property.

From a constructive, predicative point of view there are two differences that make this construction work for the Zariski lattice. Predicatively, the inclusion of basic opens into the Zariski lattice is one of small categories, while the inclusion into the classical spectrum is not. Furthermore, since we are only concerned with sheaves on a distributive lattice and not on a general locale or topological space, we only have to consider finite covers. This allows for a predicative proof that the right Kan extension preserves sheaves on lattices. From a classical point of view this is not really a restriction as $\mathsf{Spec}\, R$ is always a *coherent* space. As a result, sheaves on $\mathsf{Spec}\, R$ are in bijection to (finitary) sheaves on the Zariski lattice. For more details see e.g. Johnstone's "Stone Spaces" [18] and Section 6.1 of this paper.

Regardless of whether one formalizes the classical or constructive definitions, the main bottlenecks of the formalization are already found at the level of basic opens. First and foremost, there is the well-definedness problem described above. The second bottleneck is proving that the structure sheaf actually is a sheaf on basic opens. In fact, the problem with the textbook proof of the sheaf property is the well-definedness problem in disguise. Those two points were exactly where the most prominent formalization of schemes [4] in Lean's mathlib [26] encountered problems. In this paper, we show that with the help of univalence it is in fact possible to overcome the issues of well-definedness and formalize the structure sheaf directly on basic opens and prove its sheaf property. Even though we work in the constructive, predicative setting using the Zariski lattice, the techniques used to overcome the problems on the level of basic opens should be applicable to a classical formalization in type theory with univalence and classical axioms added. The key insight is that localizations are not just commutative rings, but also commutative algebras over R. In R-algebras, isomorphisms, and thus also paths, between two localizations are unique, which ensures well-definedness of the structure sheaf.

As mentioned above, our work is completely formalized[1] in Cubical Agda, an extension of the Agda proof assistant [31] based on the cubical type theory of [7, 8] with fully constructive support of the univalence axiom and higher inductive types (HITs). However, nothing relies crucially on cubical features, or on univalence and eliminators applied to higher constructors of HITs computing definitionally, in our formalization. The only HoTT/UF features that we rely on are univalence and set quotients (from which propositional truncation follows). It would hence be possible to perform the formalization in a system implementing Book HoTT [33] or in UniMath [36]. Our work is thus in line with the aim of Voevodsky's Foundations library [39] of developing a library of constructive set-level mathematics based on Univalent Foundations.

Contributions

As mentioned above, the formalization presented in this paper generally follows the constructive, lattice-based approach of [10]. However, a number of design choices had to be made to ensure predicativity of our formalization and to enable us to formally prove the well-definedness of the structure sheaf. As a result some definitions and proofs deviate from the presentation in [10]. The main design choices and contributions of the paper and formalization can be summarized under the following topics:

- **Commutative algebra:** our formalization of localizations of commutative rings in Section 3.1 closely follows Atiyah and MacDonald's classic textbook [2], which works very well for our constructive approach. However, giving a predicative definition of the Zariski lattice that does not increase universe levels was more intricate. To this end, Section 3.2 contains a construction that refines the ideal-based description of [10] using ideas of Español [14].

[1] All results discussed are integrated in the agda/cubical library and are summarized in: https://github.com/agda/cubical/blob/310a0956bb45ea49a5f0aede0e10245292ae41e0/Cubical/Papers/AffineSchemes.agda
This is a permalink to the library at the time of writing, which type-checks with Agda version 2.6.3.
A clickable rendered version that might be subject to change can be found here:
https://agda.github.io/cubical/Cubical.Papers.AffineSchemes.html

- **Category theory:** in Section 4 we present a formal notion of sheaf on a distributive lattice that closely follows [10]. However, in [10] presheaves are extended from a basis of a distributive lattice to the whole lattice in a somewhat non-standard finitary way. This is to ensure predicativity, but it actually causes problems when working in a univalent setting. We found that the point-wise right Kan extension of presheaves, as e.g. presented in MacLane's classic textbook [23], works just fine even in the constructive and predicative setting. We then give a proof that the Kan extension preserves the sheaf property. This can be seen as the main step towards a constructive and predicative "comparison lemma" that gives an equivalence of categories between sheaves on a lattice and sheaves on a basis of the lattice.
- **Constructive affine schemes:** in Section 5 we construct the structure sheaf on basic opens and extend it to the Zariski lattice. We give general heuristics for constructing presheaves (valued in R-algebras) on subsets defined using propositional truncation. The well-definedness of the presheaves thus constructed follows from univalence. The structure sheaf is a special instance of this construction with the basic opens seen as a subset of the Zariski lattice. Proving the sheaf property on basic opens can then be reduced to standard commutative algebra, again by using univalence in a way that does not require to extract the isomorphisms underlying the applications of univalence.

2 Background

Here we give the necessary background for the paper. We first sketch the constructive approach to schemes of [10]. We then continue with an introduction to the concepts of `Cubical Agda` needed for the paper.

2.1 Affine schemes constructively

Recall that, classically, the spectrum of a commutative ring R is the set of its prime ideals $\mathsf{Spec}\, R = \{\mathfrak{p} \subseteq R \mid \mathfrak{p} \text{ prime}\}$ equipped with the Zariski topology. The open sets of this topology are generated by basic opens $D(f) = \{\mathfrak{p} \mid f \notin \mathfrak{p}\}$ for $f \in R$. Constructively, there are two issues with this. First, the notion of prime ideal is not really well-behaved. One of the main reasons for this is that the central notion of *localizing* at a prime ideal $\mathfrak{p}$ actually uses the set-theoretic complement $R \setminus \mathfrak{p}$, which does not work well constructively without additional decidability assumptions.[2] To remedy this, one can define the notion of a *prime filter* on R and check that classically those are exactly the complements of prime ideals.

The second issue concerns the point-set definition of a topological space itself. For a constructive development of algebraic geometry it is preferable to avoid this definition and instead characterize the *locale* of open sets of $\mathsf{Spec}\, R$ in a direct, point-free way. This can be done by observing that the closed sets of the Zariski topology admit a direct algebraic characterization. Every closed set is of the form $V(\mathfrak{a}) = \{\mathfrak{p} \mid \mathfrak{a} \subseteq \mathfrak{p}\}$, where $\mathfrak{a}$ is a *radical ideal.* An ideal $\mathfrak{a} \subseteq R$ is radical if $\mathfrak{a} = \sqrt{\mathfrak{a}}$, where

$$\sqrt{\mathfrak{a}} = \left\{ x \in R \mid \exists n > 0 : x^n \in \mathfrak{a} \right\}$$

The *locale of Zariski opens* can thus be characterized by the set of radical ideals of R. The join and meet operation can be defined using addition and multiplication of ideals.

[2] See e.g. the discussion by Mines, Richman and Ruitenberg in their standard textbook on constructive algebra [27, Section III.3].

From a predicative viewpoint this is still unsatisfactory. Predicatively, the ideals of a ring form a proper class and consequently the Zariski locale is not a set in such a setting. However, by restricting to the *lattice of compact open sets* of the Zariski topology these size issues can be avoided.[3] Classically, the objects of this lattice are finite unions of basic opens $D(f_1) \cup \cdots \cup D(f_n)$ and the join and meet operation are just union $\cup$ and intersection $\cap$. Note that for the meet this only works because basic opens are closed under intersections, i.e. we have $D(f) \cap D(g) = D(fg)$ for any $f, g \in R$.

As with the locale of Zariski opens, this so-called *Zariski lattice* $\mathcal{L}_R$ of a commutative ring R can be described in a point-free way. This was first done by Joyal [19], using the observation that the Zariski lattice has a certain universal property. The lattice itself can be defined as the free distributive lattice generated by *formal symbols* $D(f)$, $f \in R$, satisfying the following relations:

$$D(1) = \top \quad \text{and} \quad D(0) = \bot \tag{1}$$

$$\forall f, g \in R : \; D(fg) = D(f) \wedge D(g) \tag{2}$$

$$\forall f, g \in R : \; D(f+g) \leq D(f) \vee D(g) \tag{3}$$

The induced map $D : R \to \mathcal{L}_R$ is universal in the following sense: for any distributive lattice L and *support* map $d : R \to L$, i.e. any map d such that conditions (1)-(3) above hold for d (in place of D), there is a unique lattice homomorphism $\varphi : \mathcal{L}_R \to L$ such that the following commutes

$$\begin{tikzcd} & R \arrow[dl, "D"'] \arrow[dr, "d"] & \\ \mathcal{L}_R \arrow[rr, dashed, "\exists! \, \varphi"'] & & L \end{tikzcd}$$

Using the correspondence of Zariski opens with radical ideals, the elements of $\mathcal{L}_R$ can also be described as the *radicals of finitely generated ideals.* For two finitely generated ideals $\mathfrak{a}, \mathfrak{b} \subseteq R$, the join and meet of the radicals are then given by

$$\sqrt{\mathfrak{a}} \vee \sqrt{\mathfrak{b}} = \sqrt{\mathfrak{a} + \mathfrak{b}} \qquad \text{and} \qquad \sqrt{\mathfrak{a}} \wedge \sqrt{\mathfrak{b}} = \sqrt{\mathfrak{a}\mathfrak{b}}$$

using the fact that addition and multiplication of two finitely generated ideals is again finitely generated. The support $D : R \to \mathcal{L}_R$ maps $f \in R$ to the radical of the principal ideal $\sqrt{\langle f \rangle}$ and for any support $d : R \to L$, the unique morphism $\varphi : \mathcal{L}_R \to L$ is given by

$$\varphi\big(\sqrt{\langle f_1, \ldots, f_n \rangle}\,\big) \;=\; d(f_1) \vee \cdots \vee d(f_n)$$

In Section 3.2, we will show how to formalize this Zariski lattice of radicals of finitely generated ideals and prove its universal property while avoiding size issues.

The lattice theoretic approach does require a notion of a sheaf on a distributive lattice. Recall that a sheaf on a topological space X is just a sheaf on the locale of open sets of X. By restricting the definition of sheaf on a locale to finite covers one obtains sheaves on a distributive lattice. This means that for any distributive lattice L, a presheaf $\mathcal{F} : L^{op} \to \mathcal{C}$, valued e.g. in commutative rings (i.e. $\mathcal{C} = \mathsf{CommRing}$), is a sheaf if for all $x_1, \ldots, x_n \in L$ the following is an equalizer diagram

[3] Through a more careful analysis one might be able to define the structure sheaf on the large Zariski locale in predicative univalent foundations, as long as one uses a small type of basic opens. See the recent work by de Jong and Hötzel Escardó [11] and by Tosun and Hötzel Escardó [32] for results of this kind. For the development of constructive and predicative scheme theory however, it seems certainly advantageous to work with the small Zariski lattice.

$$\mathcal{F}\Big(\bigvee_{i=1}^{n} x_i\Big) \to \prod_{i=1}^{n} \mathcal{F}(x_i) \rightrightarrows \prod_{i<j} \mathcal{F}(x_i \wedge x_j)$$

A basis of a distributive lattice is a subset $B \subseteq L$ containing $\top$ and closed under meets, such that for any $x \in L$ there exists a finite list $b_1, \dots, b_n \in B$ such that $x = \bigvee_{i=1}^{n} b_i$. In Section 4, we describe how to obtain sheaves on L from sheaves on B. This works analogous to the special case of the so-called *comparison lemma* for topological spaces. For the structure sheaf, the idea is to map $D(f)$ to the ring $R[1/f]$, the *localization of R away from f*. Recall that for a subset $S \subseteq R$ containing 1 and being closed under multiplication, the localization $S^{-1}R$ is defined as the ring of fractions r/s where $r \in R$ and $s \in S$. Equality of fractions is given by

$$\frac{r_1}{s_1} = \frac{r_2}{s_2} \quad \text{iff} \quad \exists u \in S : \; u(r_1 s_2 - r_2 s_1) = 0$$

$R[1/f]$ is defined by localizing with $S = \{1, f, f^2, f^3, \dots\}$. Its elements are thus fractions r/f^n where the denominator is a power of f and equality can be rephrased as

$$\frac{r}{f^n} = \frac{r'}{f^m} \quad \text{iff} \quad \exists k \in \mathbb{N} : \; f^{k+m} r = f^{k+n} r'$$

Verifying that the presheaf defined by sending $D(f)$ to $R[1/f]$ is indeed a sheaf on the basis $\mathcal{B}_R \subseteq \mathcal{L}_R$ of basic opens proceeds the same way in any constructive or classical account. As indicated in the introduction, there are some issues to be overcome when formalizing the construction of the structure sheaf. In this paper we discuss what a solution to these problems can look like in a univalent setting.[4]

2.2 Set-level univalent mathematics in Cubical Agda

We will now briefly discuss the concepts needed from `Cubical Agda` for this paper, for more details see [35]. Our notation is inspired by `Agda` syntax and the `agda/cubical` library, but we have taken some liberties when typesetting, e.g. shortening notations and omitting some projections and universe levels whenever possible. We write Type ℓ for universes (at level ℓ) and $\Sigma[\ x \in A\ ]\ B(x)$ for dependent pair types over a family $B : A \to$ Type ℓ. The major difference when working in `Cubical Agda` compared to vanilla `Agda` or Book HoTT is that the primary identity type is changed from Martin-Löf's inductive construction [25] to a primitive *path*-type. The identification $x \equiv y$ is captured by Path $A\,x\,y$, the type of functions $p : \mathsf{I} \to A$, where I is a primitive interval type, restricting definitionally to x and y at the endpoints i0 and i1 of I. `Cubical Agda` also has a dependent path type, PathP. Given a line of types $B : \mathsf{I} \to$ Type, which we may think of as $B(\mathsf{i0}) \equiv B(\mathsf{i1})$, and $x : B(\mathsf{i0})$, $y : B(\mathsf{i1})$, the type PathP $B\ x\ y$ expresses that x and y may be identified relative to B. The regular path type _≡_ is, by definition, PathP $(\lambda\ i \to A)$, i.e. the special case of a constant line of types.

`Cubical Agda` also comes with a function ua : $A \simeq B \to A \equiv B$ which promotes equivalences (or isomorphisms) of types to paths between these types. The fact that this map is an equivalence itself is a way to formulate Voevodsky's univalence axiom. A reasonable question to ask in a univalent setting is whether an equivalence of types can be promoted to

[4] A solution that is e.g. taken in [10], is to map $D(f)$ to $S_f^{-1}R$, the ring of fractions whose denominators are elements of the *saturation* $S_f = \{g \mid D(f) \subseteq D(g)\}$. It is immediate to see that if $D(f) = D(g)$, then $S_f^{-1}R = S_g^{-1}R$, but it is not as natural to work with these rings. Usually, one still wants to appeal to the "canonical isomorphism" between $R[1/f]$ and $S_f^{-1}R$, as in e.g. [13, Sect. 1.3].

an equality of structured types, such as groups or rings. The *Structure Identity Principle (SIP)* [33, Sect. 9.8] is an informal principle which attempts to answer this: given two structured types (A, S_A) and (B, S_B) and an equivalence of underlying types $A \simeq B$ which is a homomorphisms with respect to the structure in question, we get a path of structured types $(A, S_A) \equiv (B, S_B)$. For instance, an isomorphism of rings R and S induces a path $R \equiv S$. This has been implemented in `agda/cubical` using the cubical SIP of Angiuli, Cavallo, Mörtberg and Zeuner [1]. For this paper we will use sip to denote the function that turns isomorphisms of commutative rings or R-algebras (over a ring R) into paths.

Univalence refutes *Uniqueness of Identity Proofs (UIP)*, or Streicher's axiom K [30], because it produces equality proofs in Type that are not equal [33, Ex. 3.1.9]. In the presence of univalence, it is therefore important to keep track of which types satisfy UIP or related principles expressing the complexity of a type's equality relation. In the terminology of HoTT/UF, a type satisfying UIP is called an *h-set* (*homotopy set*, henceforth simply *set*), while a type whose elements are all equal is called an *h-proposition* (henceforth *proposition*).

Another very important concept in HoTT/UF is that of *contractible* types, i.e. types with exactly one element:

```
isContr : Type → Type
isContr A = Σ[ x ∈ A ] ((y : A) → x ≡ y)
```

We can characterize propositions as types whose equality types are contractible, just as sets are types whose equality types are propositions. Thus contractible types, propositions, and sets serve as the bottom three layers of an infinite hierarchy of types introduced by Voevodsky, known as *h-levels* [37] or *n-types* [33]. This paper is about set-level mathematics, so we are mainly interested in these 3 bottom layers. However, univalence implies that collections of set-level structures (e.g. the collection of all commutative rings or R-algebras) are one level higher than sets. Types at this level are called *h-groupoids* (heceforth *groupoids*) and will be the only types of h-level higher than 2 in the paper. We write isProp A to say that A is a proposition and isSet A to say that A is a set. The "universe of propositions" hProp ℓ is defined as Σ[A ∈ Type ℓ] (isProp A), and if isSet A, we call functions $S : A \to$ hProp ℓ a *subset* of A. For $a : A$ we denote by $a \in S$ the type of proofs that a is actually in S. It is often convenient to identify the subset S with Σ[a ∈ A] $(a \in S)$, which can be seen as a sub-type of A. With some abuse of notation we will not distinguish between subsets as functions and the corresponding Σ-type. We thus write $a : S$ for elements of S when the proof of a belonging to S can be ignored.

Another concept from HoTT/UF which `Cubical Agda` supports are higher inductive types (HITs). These allow us to define many important operations on types, such as truncations. For instance, the propositional truncation is defined by:

```
data ∥_∥ (A : Type ℓ) : Type ℓ where
  ∣_∣ : A → ∥ A ∥
  squash : isProp ∥ A ∥
```

This HIT takes a type A and forces it to be a proposition. This is a very important construction for capturing existential quantification in HoTT/UF:

```
∃[ x ∈ A ] P(x) = ∥ Σ[ x ∈ A ] P(x) ∥
```

In this paper, we follow the HoTT Book terminology and say that x *merely* exists when it is existentially quantified. Note that the propositional truncation in the definition is crucial. In HoTT/UF, Σ A P without the truncation is interpreted as the total space of P, which may be highly non-trivial. For example, a subset B of a lattice L is a basis if

$$\forall\,(x : L) \to \exists[\, b_1, \dots, b_n \in B \,]\ (\textstyle\bigvee_{i=1}^{n} b_i \equiv x)$$

Here the propositional truncation is necessary. We will see this in Section 3.2, when proving that the basic opens form a basis of $\mathcal{L}_R$.

The main HIT that we use in this paper is the *set quotient*, which quotients a type by an arbitrary relation, yielding a set. It has three constructors: [_], which includes elements of the underlying type, eq/, which equates all pairs of related elements, and squash/, which ensures that the resulting type is a set:

```
data _/_ (A : Type ℓ) (R : A → A → Type ℓ) : Type ℓ where
  [_] : (a : A) → A / R
  eq/ : (a b : A) → (r : R a b) → [ a ] ≡ [ b ]
  squash/ : isSet (A / R)
```

We can write functions out of A / R by pattern-matching; this amounts to writing a function out of A (the clause for [_]) which sends R-related elements of A to equal results (the clause for eq/), such that the image of the function is a set (the clause for squash/). Set quotients and propositional truncations have in common that the resulting type will be of a fixed h-level and this makes it very hard to map into types of higher h-levels. In fact, the higher the h-level of the target type, the more complicated the coherence conditions that need to be proved. We will see an example of this in Section 5.

3 Commutative algebra

In this section, we first discuss our formalization of localizations of rings, followed by the definition of the Zariski lattice. These objects can be described by universal properties, but may also be concretely implemented as set quotients. One of the guiding principles of this project was to work with concrete implementations and mainly use universal properties to construct equivalences and paths (via the SIP). As a result the formalization follows the usual informal treatment in the commutative algebra literature quite closely.

3.1 Localizations

Our formalization of localizations of commutative rings follows the classic textbook of Atiyah and MacDonald [2], with our main result being a path version of [2, Cor. 3.2]. Note that the definition of localization is actually the same in classical and constructive algebra.[5] For the remainder of this paper we will only consider commutative rings with a multiplicative unit (denoted by 1). Let R be such a ring and S a subset of R that contains 1 and is closed under multiplication. The formalization of localization is then straightforward:

```
S⁻¹R : Type
S⁻¹R = (R × S) / _≈_
  where
  _≈_ : R × S → R × S → Type
  (r₁ , s₁ , _) ≈ (r₂ , s₂ , _) = Σ[ (u , _) ∈ S ] (u · r₁ · s₂ ≡ u · r₂ · s₁)
```

The underscores in the definition of ≈ correspond to the proofs that s_1, s_2 and s are elements of S respectively. As these are unimportant to the definition of ≈, we can safely omit them.

[5] Compare [2] with e.g. the books by Lombardi and Quitté [22] or Mines, Richman and Ruitenburg [27].

▶ Remark 1. It might be surprising that we define $\approx$ using a Σ and not an $\exists$ (as is done e.g. in [39]). However, it turns out that it does not matter whether one quotients by the truncated relation using $\exists$ or the untruncated relation using Σ, as the resulting set-quotients will be equivalent. As we do not need to prove anything about $\approx$ except it being an equivalence relation, it is more convenient to work without the truncation.

Equipping $S^{-1}R$ with the structure of a commutative ring is laborious in `Cubical Agda`, but the proofs generally proceed as in any textbook. The same holds for the universal property. Note that for this we need the canonical homomorphism $\_/1 : R \to S^{-1}R$, mapping $r : R$ to $[\ r\ ,\ 1\ ]$, the equivalence class corresponding to $r/1$. The universal property then states that for any commutative ring A with a morphism $\varphi : R \to A$, such that for all $s : S$ we have $\varphi(s) \in A^{\times}$ (i.e. that $\varphi(s)$ is a unit in A), there is a unique morphism $\psi : S^{-1}R \to A$, such that the following commutes

$$\begin{tikzcd}
 & R \arrow[dl, "\_/1"'] \arrow[dr, "\varphi"] & \\
S^{-1}R \arrow[rr, dashed, "\exists!\ \psi"'] & & A
\end{tikzcd}$$

The key observation for the main results of this paper is that localizations are R-algebras via the canonical homomorphism $\_/1$. The type of R-algebras is equivalent to the Σ-type of a commutative ring A together with a ring homomorphism $\varphi : R \to A$. An homomorphism between R-algebras (A, φ) and (B, ψ) is just a ring homomorphism $\chi : A \to B$ together with a path $\chi \circ \varphi \equiv \psi$. The type of R-algebra homomorphisms will be denoted by $\mathrm{Hom}_R\big[(A, \varphi)\,, (B, \psi)\big]$ or just $\mathrm{Hom}_R\big[A, B\big]$ if the morphisms are clear from context.

The universal property of localization then becomes a statement about R-algebras. In HoTT/UF unique existence is defined as contractibility of Σ-types, so the universal property of the localization at S becomes: *for any R-algebra (A, φ) s.t. $\varphi(S) \subseteq A^{\times}$, the type $Hom_R\big[S^{-1}R\,, (A, \varphi)\big]$ is contractible.* Combining the proof of [2, Cor. 3.2] with the SIP, we can then prove the following:[6]

▶ **Lemma 2.** *Let A be a commutative ring with a morphism $\varphi : R \to A$ satisfying*
- $\forall (s : S) \to \varphi(s) \in A^{\times}$
- $\forall (r : R) \to \varphi(r) \equiv 0 \to \exists[\ s \in S\ ]\ (sr \equiv 0)$
- $\forall (a : A) \to \exists[\ (r\ ,\ s) \in R \times S\ ]\ (\varphi(r)\varphi(s)^{-1} \equiv a)$

From this we can construct a path $S^{-1}R \equiv A$, which is unique as a path in R-algebras.

With this result we can transport proofs about localizations to any suitable ring and morphism pair, i.e. R-algebra, satisfying the three conditions above. Below we will see a few applications of this result that will be used for formalizing constructive affine schemes. The important case for our purpose is $R[1/f]$, the localization of R *away from* f. This can be seen as inverting a single element f in R. The subset $S = \{1, f, f^2, f^3, \ldots\}$ is easily defined in `Cubical Agda` as the set of $g : R$ for which we have an inhabitant of $\exists[\ n \in \mathbb{N}\ ]\ (g \equiv f^n)$.

For the remainder of this section let $f, g : R$. By the canonical homomorphism we get an element $g/1$ in $R[1/f]$. With a bit of abuse of notation we denote the localization away from this element by $R[1/f][1/g]$. This is an R-algebra by applying the canonical morphism $\_/1$ twice. We can of course also localize away from $(f \cdot g)$, thus obtaining $R[1/fg]$. Using Lemma 2, we can construct a (unique) path between these two, which will be used for the structure sheaf. Similarly, we also get other useful paths.

[6] In `Lean`'s `mathlib` *a* localization is defined to be any ring-morphism-pair satisfying the three conditions of Lemma 2. The formulation of this predicate is attributed to Neil Strickland in [4].

▶ **Lemma 3.** *We have the following paths for both commutative rings and R-algebras:*

1. $R[1/f][1/g] \equiv R[1/fg]$
2. $R[1/f] \equiv R$, *if* $f \in R^\times$
3. $R[1/f] \equiv R[1/g]$, *if* $f/1 \in R[1/g]^\times$ *and* $g/1 \in R[1/f]^\times$

3.2 The Zariski lattice

Next, we provide a definition of the Zariski lattice that does not lead to size issues, while still being convenient to work with. We have already seen that the Zariski lattice $\mathcal{L}_R$, which classically corresponds to the compact open sets of the Zariski topology, can be described as the lattice of radicals of finitely generated ideals. The meet and join of this lattice are defined using multiplication and addition of ideals. With some elementary ideal theory this should be straightforward to formalize. Unfortunately, without any form of impredicativity, like resizing axioms, this leads to size issues.

So far we have avoided being explicit about universe levels, but in this section let ℓ be the level of the base ring R, that is, the level of the universe in which the underlying type of R lives. Being precise about universe levels, subsets of R are elements of $R \to \mathsf{hProp}\ \ell$, living in $\mathsf{Type}\ (\ell + 1)$, the next bigger universe. The type of all ideals of R, which is just the Σ-type of subsets satisfying the ideal property, is hence in $\mathsf{Type}\ (\ell + 1)$. However, for technical reasons to be discussed in the next section, we need $\mathcal{L}_R : \mathsf{Type}\ \ell$. Consequently, the definition of $\mathcal{L}_R$ must not rely on the type of all ideals of R.

To avoid this issue, we use a construction due to Español [14]. Since we are only concerned with the radicals of finitely generated ideals, we can describe $\mathcal{L}_R$ in terms of generators instead of arbitrary ideals. In particular, a list of generators $\alpha = [\alpha_0, \ldots, \alpha_n]$ with $\alpha_i : R$ corresponds to the radical of the ideal generated by the α_i. In other words, we can obtain $\mathcal{L}_R$ by quotienting the type of lists with elements in R, by the relation

$$\alpha \sim \beta \quad \Leftrightarrow \quad \left(\forall\ i \to \beta_i \in \sqrt{\langle \alpha_0, \ldots, \alpha_n \rangle}\,\right) \text{ and } \left(\forall\ i \to \alpha_i \in \sqrt{\langle \beta_0, \ldots, \beta_m \rangle}\,\right)$$

Here $\langle \alpha_0, \ldots, \alpha_n \rangle$ is the ideal generated by the α_i's. As both the type of lists and $\sim$ live in $\mathsf{Type}\ \ell$ so does their quotient $\mathcal{L}_R$. It might seem more natural to quotient by the relation

$$\alpha \approxeq \beta \quad \Leftrightarrow \quad \sqrt{\langle \alpha_0, \ldots, \alpha_n \rangle} \equiv \sqrt{\langle \beta_0, \ldots, \beta_m \rangle}$$

Unfortunately the type of paths between two such radicals is large, as for any two ideals I, J we have $I \equiv J : \mathsf{Type}\ (\ell + 1)$. Still, $\sim$ is equivalent to $\approxeq$ in the sense that we have $\alpha \sim \beta$ if and only if $\alpha \approxeq \beta$. This equivalence can then be used in proofs.

Equipping $\mathcal{L}_R$ with the distributive lattice structure requires us to introduce operations on lists that correspond to ideal addition and multiplication. For the join we can take list concatenation $\_{+\!+}\_$ as this corresponds to addition of finitely generated ideals in the sense that for any two lists α, β we have that

$$\begin{aligned} \langle\ [\alpha_0, \ldots, \alpha_n] +\!+ [\beta_0, \ldots, \beta_m]\ \rangle &\equiv \langle \alpha_0, \ldots, \alpha_n, \beta_0, \ldots, \beta_m \rangle \\ &\equiv \langle \alpha_0, \ldots, \alpha_n \rangle + \langle \beta_0, \ldots, \beta_m \rangle \end{aligned} \tag{4}$$

When checking that $\_{+\!+}\_$ defines an operation on the quotient $\mathcal{L}_R$, it suffices to check that it respects $\approxeq$, which in turn follows from (4).

For the meet of $\mathcal{L}_R$ we need to define an operation $\_\cdot\cdot\_$ on lists that corresponds to multiplication of finitely generated ideals. For two lists α, β this product $\alpha \cdot\cdot \beta$ is the list of all products of the form $\alpha_i \beta_j$. Proving the correspondence to ideal multiplication, i.e.

$$\begin{aligned}\langle\, [\alpha_0,\dots,\alpha_n]\cdot\cdot[\beta_0,\dots,\beta_m]\,\rangle &\equiv \langle \alpha_0\beta_0,\dots,\alpha_n\beta_0,\dots,\alpha_0\beta_m,\dots,\alpha_n\beta_m\rangle \\ &\equiv \langle \alpha_0,\dots,\alpha_n\rangle\cdot\langle\beta_0,\dots,\beta_m\rangle \qquad (5)\end{aligned}$$

is much more involved than (4), but gives us the well-definedness of $\_\cdot\cdot\_$ on the quotient. Proving the lattice laws also proceeds by using (4) and (5), together with the equivalence of $\sim$ and $\approx$, thus reducing these laws to special cases of standard equalities about ideal addition/multiplication and radical ideals.

Showing the universal property of $\mathcal{L}_R$ is then relatively straightforward. Note that the basic opens are defined by the map $D : R \to \mathcal{L}_R$, sending $f : R$ to $[\ [f]\ ]$, the equivalence class of the singleton list $[f]$. It then becomes straightforward to verify that for $f, g : R$

$$D(g) \le D(f) \;\Leftrightarrow\; \sqrt{\langle g\rangle} \subseteq \sqrt{\langle f\rangle} \;\Leftrightarrow\; f \in R[1/g]^{\times} \;\Leftrightarrow\; \mathsf{isContr}\Big(\mathrm{Hom}_R\big[R[1/f]\,,\,R[1/g]\,\big]\Big)$$

The last two logical equivalences hold by some standard commutative algebra and the universal property of localization.[7] The basic opens as a subset of $\mathcal{L}_R$ are defined as the function $\mathsf{BasicOpens} : \mathcal{L}_R \to \mathsf{hProp}$, sending $\mathfrak{a}$ to $\exists[\ f \in R\ ]\ (D(f) \equiv \mathfrak{a})$. In other words $\mathfrak{a} \in \mathsf{BasicOpens}$ if there merely exists an f such that $\mathfrak{a}$ equals $D(f)$. The type of basic opens is then the type $\mathcal{B}_R = \Sigma[\ \mathfrak{a} \in \mathcal{L}_R\ ]\ (\mathfrak{a} \in \mathsf{BasicOpens})$. Note that by the universal property, the only lattice morphism $\mathcal{L}_R \to \mathcal{L}_R$ commuting with D is the identity and from this it follows that for any list $\alpha = [\alpha_0,\dots,\alpha_n]$ the equivalence class $[\ \alpha\ ]$ is the finite join $\bigvee_{i=0}^{n} D(\alpha_i)$. Since being a basis is a proposition, this is enough to prove that the basic opens form a basis of $\mathcal{L}_R$.

4 Category theory

We now turn to category theory and describe the machinery needed to lift sheaves from the basis of a distributive lattice to the whole lattice. The lifting of a presheaf defined on a subset of a distributive lattice, seen as a sub-poset category, is obtained by taking the right Kan extension along the inclusion. The general theory of limits and Kan extensions in the formalization closely follows Mac Lane [23]. We will not discuss details here, but only sketch the lattice case in order to introduce notation and show where size issues enter the picture.

Note that for any category $\mathcal{C}$ and $P : \mathcal{C} \to \mathsf{hProp}$, $\mathcal{C}_P = \Sigma[\ x \in \mathcal{C}\ ]\ (x \in P)$ becomes a subcategory of $\mathcal{C}$ by taking arrows between pairs to be arrows between the first projections. The projection fst induces a fully faithful embedding of $\mathcal{C}_P$ into $\mathcal{C}$. Let us now fix a distributive lattice $L : \mathsf{Type}\ \ell$. For any $P : L \to \mathsf{hProp}\ \ell$, L_P becomes a sub-poset of L.

Let $\mathcal{C}$ be an ℓ-complete category (i.e. with limits of diagrams in $\mathsf{Type}\ \ell$). The right Kan extension then exists for any $\mathcal{C}$-valued presheaf $\mathcal{G}$ on L_P:

$$\begin{tikzcd} (L_P)^{op} \arrow[d, hook, "\mathsf{fst}"'] \arrow[dr, "\mathcal{G}"] & \\ L^{op} \arrow[r, dashed, "\mathsf{Ran}\ \mathcal{G}"'] & \mathcal{C} \end{tikzcd}$$

$$(\mathsf{Ran}\ \mathcal{G})\,(x) \;=\; \varprojlim\ \big\{\,\mathcal{G}(u) \to \mathcal{G}(v) \mid u, v : L_P \text{ s.t. } v \le u \le x\,\big\}$$

[7] As all the types above are propositions, we could also replace logical equivalence with equivalence of types $\simeq$.

Moreover, since the functor induced by fst is fully faithful, Ran $\mathcal{G}$ extends $\mathcal{G}$ in the sense that we have a natural isomorphism between $\mathcal{G}$ and $(\text{Ran } \mathcal{G}) \circ$ fst. For the structure sheaf we need to consider presheaves valued in CommRing ℓ, the category of commutative rings living in the same universe as the base ring R. This category is ℓ-complete but *not* $(\ell + 1)$-complete. It is precisely for this reason that we required $\mathcal{L}_R$ to be in Type ℓ.

The main result of this section is that taking the right Kan extension of a presheaf defined on the *basis* of a lattice preserves the sheaf property.[8] This requires a definition of sheaf on both distributive lattices and their bases suitable for formalization. For the remainder of this section we fix a basis B of L. When outlining the formalization, we defined sheaves on lattices by restricting the usual definition in terms of equalizer diagrams to finite covers. However, we can express these equalizers as finite limits over diagrams of a certain shape.[9] This approach is also taken by Coquand, Lombardi and Schuster in [10]. We decided to follow it as it allows one to work with special data types for the shapes of the diagrams involved, which is convenient in the formalization.

▶ **Definition 4** (Sheaf diagram shapes). *The category of the sheaf diagram shape for covers of size n, has as objects indices i, where $1 \leq i \leq n$, or pairs of indices (i, j), where $1 \leq i < j \leq n$. Arrows are either identity arrows or inclusions of singleton indices from the left $i \mapsto (i, j)$ or right $j \mapsto (i, j)$.*

In Agda the objects and arrows can be described as the terms of the following data types:

```
data DLShfDiagOb (n : ℕ) : Type where
  sing : Fin n → DLShfDiagOb n
  pair : (i j : Fin n) → i < j → DLShfDiagOb n

data DLShfDiagHom (n : ℕ) : DLShfDiagOb n → DLShfDiagOb n → Type where
  idAr : {x : DLShfDiagOb n} → DLShfDiagHom n x x
  singPairL : {i j : Fin n} {p : i < j} → DLShfDiagHom n (sing i) (pair i j p)
  singPairR : {i j : Fin n} {p : i < j} → DLShfDiagHom n (sing j) (pair i j p)
```

Here Fin n is the finite type of n elements from 1 to n. Composition is easily defined by case analysis as it is not possible to compose two non-identity arrows and the laws then follow directly. We denote the resulting category by DLShfDiagCat n.

▶ Remark 5. In order for this to define a category in HoTT/UF we have to prove that the hom-types are sets, i.e. that for x, y : DLShfDiagOb n we have isSet (DLShfDiagHom n x y). This follows from a retraction argument using the encode-decode method [33].

Given a list of elements $\alpha = [\alpha_1, \ldots, \alpha_n]$ with $\alpha_i : L$, we get a corresponding diagram in the form of a functor DLShfDiagCat $n \to L^{op}$ sending the singleton index i to α_i and (i, j) to $\alpha_i \wedge \alpha_j$. We call this the *diagram associated to* α. Furthermore, let $\mathcal{F} : L^{op} \to \mathcal{C}$ be a presheaf, we then have a diagram DLShfDiagCat $n \to \mathcal{C}$, obtained by composing the diagram associated to α with $\mathcal{F}$. We call this the *$\mathcal{F}$-diagram associated to α*.

The join $\bigvee_{i=1}^{n} \alpha_i$ induces a cone over the diagram associated to α and it is in fact a limiting cone because limits are least upper bounds in the opposite of a poset category. A presheaf on L is a sheaf if it preserves these limits:

[8] In fact the right Kan extension (as opposed to left Kan) establishes an equivalence of categories between sheaves on a lattice L and sheaves on a basis B of L, with its inverse being restriction to B. This is the special case of the so-called *comparison lemma* for distributive lattices.

[9] See e.g. Mac Lane [23, Thm. V.2.1].

▶ **Definition 6** (Sheaves on a distributive lattice). *We say that $\mathcal{F}$ is a sheaf on the distributive lattice L, if for all lists $\alpha = [\alpha_1, \ldots, \alpha_n]$ with $\alpha_i : L$ the induced cone of $\mathcal{F}\big(\bigvee_{i=1}^n \alpha_i\big)$ over the $\mathcal{F}$-diagram associated to α is a limiting cone. In other words $\mathcal{F}\big(\bigvee_{i=1}^n \alpha_i\big)$ is the limit of the diagram*

$$\begin{tikzcd}
 & \mathcal{F}\big(\bigvee_{i=1}^n \alpha_i\big) \arrow[dl] \arrow[d] \arrow[dr] & \\
\mathcal{F}(\alpha_i) \arrow[r] & \mathcal{F}(\alpha_i \wedge \alpha_j) & \mathcal{F}(\alpha_j) \arrow[l]
\end{tikzcd} \qquad \textit{for all} \quad 1 \le i < j \le n.$$

We now turn our attention to the corresponding notion for the basis B. Let $\mathcal{G} : B^{op} \to \mathcal{C}$ be a presheaf. For a list $\alpha = [\alpha_1, \ldots, \alpha_n]$ with $\alpha_i : B$, we have a diagram DLShfDiagCat $n \to \mathcal{C}$, which is obtained by composing the diagram associated to α with $\mathcal{G}$. We call this the *$\mathcal{G}$-diagram associated to α*. As B is in general not closed under finite joins, the definition of a basis-sheaf below has an extra condition, saying that limits of the associated diagrams are only preserved if they exist.

▶ **Definition 7** (Sheaves on a basis of a distributive lattice). *We say that $\mathcal{G}$ is a sheaf on the basis B of a distributive lattice, if for all $\alpha = [\alpha_1, \ldots, \alpha_n]$ with $\alpha_i : B$, such that $\bigvee_{i=1}^n \alpha_i$ is in B, the induced cone of $\mathcal{G}\big(\bigvee_{i=1}^n \alpha_i\big)$ over the $\mathcal{G}$-diagram associated to α is a limiting cone.*

The following lemma only holds for sheaves on the whole lattice, since it requires closure under finite joins.

▶ **Lemma 8.** *Let $\mathcal{F} : L^{op} \to \mathcal{C}$, then $\mathcal{F}$ is sheaf if and only if $\mathcal{F}(\bot)$ is terminal in $\mathcal{C}$ and for all $x, y : L$ the following is a pullback square*

$$\begin{tikzcd}
\mathcal{F}(x \vee y) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathcal{F}(x) \arrow[d] \\
\mathcal{F}(y) \arrow[r] & \mathcal{F}(x \wedge y)
\end{tikzcd}$$

Proof. We start by observing that Definition 6 also applies to the empty list []. The join over [] is just $\bot$ and the associated diagram is the "empty" diagram. So if $\mathcal{F}$ is a sheaf then $\mathcal{F}(\bot)$ is terminal. Furthermore, the pullback squares are exactly the sheaf condition for two element lists. This concludes the "only if" direction.

For the other direction, we proceed by induction on the length n. The base case $n = 0$ follows from $\mathcal{F}(\bot)$ being terminal. For the inductive step take a list $\alpha_1, \ldots, \alpha_n : L$ of length n. By assumption the following is a pullback square

$$\begin{tikzcd}
\mathcal{F}\big(\bigvee_{i=1}^n \alpha_i\big) \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] & \mathcal{F}\big(\bigvee_{i=2}^n \alpha_i\big) \arrow[d] \\
\mathcal{F}(\alpha_1) \arrow[r] & \mathcal{F}\big(\bigvee_{i=2}^n (\alpha_1 \wedge \alpha_i)\big)
\end{tikzcd}$$

Now both lists $\alpha_1, \ldots, \alpha_n$ and $\alpha_1 \wedge \alpha_1, \ldots, \alpha_1 \wedge \alpha_n$ are of length $n - 1$. By applying the induction hypothesis to both, one can easily check that $\mathcal{F}\big(\bigvee_{i=1}^n \alpha_i\big)$ is the desired limit. ◀

This alternative characterization can be used to prove our "comparison lemma" for distributive lattices. For the remainder of this section, let $\mathcal{G} : B^{op} \to \mathcal{C}$ be a sheaf on the basis B. The key observation is the following technical lemma.

▶ **Lemma 9.** *For any list of elements $\alpha_1, \dots, \alpha_k : B$, we have that*[10]

$$\big(\mathsf{Ran}\ \mathcal{G}\big)\big(\textstyle\bigvee_{i=1}^{k} \alpha_i\big) \;\cong\; \varprojlim\ \big\{\, \mathcal{G}(\alpha_i) \to \mathcal{G}(\alpha_i \wedge \alpha_j) \leftarrow \mathcal{G}(\alpha_j) \mid 1 \le i < j \le k \,\big\} \tag{6}$$

Proof sketch. By definition we have

$$\big(\mathsf{Ran}\ \mathcal{G}\big)\big(\textstyle\bigvee_{i=1}^{k} \alpha_i\big) \;=\; \varprojlim\ \big\{\, \mathcal{G}(u) \to \mathcal{G}(v) \mid u, v : B \text{ s.t. } v \le u \le \bigvee_{i=1}^{k} \alpha_i \,\big\}$$

This immediately gives us the map from left to right, since we can restrict the defining diagram of $\big(\mathsf{Ran}\ \mathcal{G}\big)\big(\bigvee_{i=1}^{k} \alpha_i\big)$ to the $\mathcal{G}$-diagram associated to α.

For the inverse map we have to show that given any $X : \mathcal{C}$ with a cone based at X over the $\mathcal{G}$-diagram associated to α, we can extend this to a cone based at X over the defining diagram of $\big(\mathsf{Ran}\ \mathcal{G}\big)\big(\bigvee_{i=1}^{k} \alpha_i\big)$. Assume we have $X : \mathcal{C}$ with such a cone and let $u : B$ such that $u \le \bigvee_{i=1}^{k} \alpha_i$. Then $\bigvee_{i=1}^{k} (u \wedge \alpha_i) \equiv u$ and hence $\bigvee_{i=1}^{k} (u \wedge \alpha_i)$ is in B. This means that we can apply the assumption that $\mathcal{G}$ is a sheaf to this join. By substituting along this path, we can see $\mathcal{G}(u)$ as the limit of the $\mathcal{G}$-diagram associated to the $u \wedge \alpha_i$'s. By composing with restrictions we get a cone based at X over the $\mathcal{G}$-diagram associated to the $u \wedge \alpha_i$'s, and thus an arrow $X \to \mathcal{G}(u)$. It is not hard to show that this is functorial in u, which gives us the desired inverse arrow. The proof that the two maps are mutually inverse, is quite cumbersome and we will omit it here. ◀

The proof of the following theorem is the most technical of the entire formalization, so again we only give an outline.

▶ **Theorem 10.** $\mathsf{Ran}\ \mathcal{G}$ *is a sheaf on the distributive lattice L.*

Proof sketch. It suffices to check the terminal and pullback condition of Lemma 8. We will restrict our attention to the pullback case here. Let $x, y : L$ and note that, as being a pullback square is a proposition, we can take covers $x \equiv \bigvee_{i=1}^{n} \beta_i$ and $y \equiv \bigvee_{i=1}^{m} \gamma_i$ by base elements, i.e. $\beta_i, \gamma_j : B$ for all i and j. Substituting these covers for x and y, we have to prove the following: given $X : \mathcal{C}$ and arrows f and g such that the outer square in the diagram below commutes, then there is a unique arrow h making the whole diagram commute:

$$\begin{tikzcd}
X \arrow[rd, dashed, "\exists!\ h"] \arrow[rrd, bend left, "f"] \arrow[rdd, bend right, "g"'] & & \\
& \big(\mathsf{Ran}\ \mathcal{G}\big)\Big(\bigvee_{i=1}^{n+m} (\beta \mathbin{+\!+} \gamma)_i\Big) \arrow[r] \arrow[d] & \big(\mathsf{Ran}\ \mathcal{G}\big)\Big(\bigvee_{i=1}^{n} \beta_i\Big) \arrow[d] \\
& \big(\mathsf{Ran}\ \mathcal{G}\big)\Big(\bigvee_{i=1}^{m} \gamma_i\Big) \arrow[r] & \big(\mathsf{Ran}\ \mathcal{G}\big)\Big(\big(\bigvee_{i=1}^{n} \beta_i\big) \wedge \big(\bigvee_{i=1}^{m} \gamma_i\big)\Big)
\end{tikzcd} \tag{7}$$

Here $(\beta \mathbin{+\!+} \gamma)$ is the list-concatenation of β and γ. Applying Lemma 9 to $(\beta \mathbin{+\!+} \gamma)$, we get such an arrow h from a cone based at X over the diagram

$$\big\{\, \mathcal{G}\big((\beta \mathbin{+\!+} \gamma)_i\big) \to \mathcal{G}\big((\beta \mathbin{+\!+} \gamma)_i \wedge (\beta \mathbin{+\!+} \gamma)_j\big) \leftarrow \mathcal{G}\big((\beta \mathbin{+\!+} \gamma)_j\big) \mid 1 \le i < j \le n + m \,\big\}$$

[10] This is actually how the extension $\big(\mathsf{Ran}\ \mathcal{G}\big)$ is defined in [10]. However, in general we cannot use concrete covers of arbitrary elements of L by base elements to construct a functor into $\mathcal{C}$ if its h-level is unknown.

To construct such a cone, we apply Lemma 9 to both β and γ and precompose the resulting limiting cones with f and g respectively. This gives us two cones based at X, one over the $\mathcal{G}$-diagram associated to β and the other one over the $\mathcal{G}$-diagram associated to γ. Note that the two cones are compatible in the following sense: for all $1 \leq i \leq n$ and $1 \leq j \leq m$ the following square commutes

$$\begin{tikzcd}
X \arrow[r] \arrow[d] & \mathcal{G}(\beta_i) \arrow[d] \\
\mathcal{G}(\gamma_j) \arrow[r] & \mathcal{G}(\beta_i \wedge \gamma_j)
\end{tikzcd}$$

This is because the outer square in diagram (7) commutes and it is sufficient to construct a cone based at X over the $\mathcal{G}$-diagram associated to $(\beta +\!\!+ \gamma)$.

Note that the induced h is the unique cone morphism between the cone thus constructed and the limiting cone obtained from applying Lemma 9 to $(\beta +\!\!+ \gamma)$. Moreover, f and g are the unique cone morphisms between their respective precomposition-cones based at X and the limiting cones obtained from applying Lemma 9 to β and γ respectively. From this it follows by a cumbersome diagram chase that h is the unique morphism making the two triangles in diagram (7) commute. ◀

Formalizing the gaps in the above proof sketches is quite tedious and uses involved transports. We refer the interested reader to the formalization.

5 The structure sheaf

We now have all the ingredients needed to formalize the structure sheaf. The basic opens $\mathcal{B}_R$ form a basis of $\mathcal{L}_R$ and we have seen in the previous section how sheaves can be extended along the embedding fst : $\mathcal{B}_R \to \mathcal{L}_R$. What should the structure sheaf on $\mathcal{B}_R$ then look like? Focusing on the underlying presheaf and its action on objects for now, we need a function $\mathcal{B}_R \to \mathsf{CommRing}\ \ell$, which upon unfolding the definition of $\mathcal{B}_R$ becomes

$$\Big(\Sigma[\ \mathfrak{a} \in \mathcal{L}_R\ ]\ \underbrace{\exists[\ f \in R\ ]\ (D(f) \equiv \mathfrak{a})}_{\text{prop. trunc.}}\Big) \longrightarrow \underbrace{\mathsf{CommRing}\ \ell}_{\text{groupoid}}$$

Since membership in $\mathcal{B}_R$ is defined as a mere existence condition using propositional truncation, we can only specify the behavior of the structure sheaf in the case where we are given a point constructor of this truncation. If $\mathfrak{a} : \mathcal{L}_R$ is a basic open, such an element of the truncation consists of an element $f : R$ and a path $p : D(f) \equiv \mathfrak{a}$. In this case we know that the structure sheaf should send $(\ \mathfrak{a}\ ,\ |\ f\ ,\ p\ |\ )$ to $R[1/f]$. If the goal type were a proposition, this would be enough to specify a function. However, the type of commutative rings is a groupoid, requiring us to construct some non-trivial higher coherences.

To circumvent this problem we use the observation that the localizations are actually R-algebras and that we could regard the structure sheaf as taking values in R-algebras. What is usually called the structure sheaf in the literature is this R-algebra-valued sheaf composed with the forgetful functor to commutative rings. In other words, the structure sheaf factors through the forgetful functor from R-algebras to commutative rings. The single reason why the situation is more well-behaved in R-algebras is the fact that

$$D(g) \leq D(f) \quad \Longleftrightarrow \quad \mathsf{isContr}\Big(\mathrm{Hom}_R\big[R[1/f]\ ,\ R[1/g]\ \big]\Big)$$

Contractibility is a powerful concept in HoTT/UF and we will show how this can be used to solve the coherence issues of the structure sheaf and gives rise to a reduction argument for the sheaf property. We start with two lemmas for general constructions involving propositional truncations and R-algebras. Note that these results are pretty much tailored to the situation of the structure sheaf, but should also hold for other univalent categories, which are always groupoids and even sets if they are posetal [33, Lemma 9.1.9, Ex. 9.1.14]. With a bit of abuse of notation we will use R-Alg to denote both the type and the category of R-algebras.

▶ **Lemma 11.** *Let $X : \mathit{Type}$ and $\mathcal{F} : X \to R\text{-}\mathit{Alg}$. Assume further that for $x, y : X$ we have an isomorphism of R-algebras $\varphi_{xy} : \mathcal{F}(x) \cong \mathcal{F}(y)$ such that for $x, y, z : X$ we have a path $\varphi_{xz} \equiv \varphi_{yz} \circ \varphi_{xy}$. Then we can construct a map $\|\mathcal{F}\| : \| X \| \to R\text{-}\mathit{Alg}$ such that for $x : X$ we have $\|\mathcal{F}\|(\,|\ x\ |\,) = \mathcal{F}(x)$ definitionally.*

Proof. Since R-Alg is a groupoid, we can apply a result by Kraus [20, Prop. 2.3]. In order to construct $\|\mathcal{F}\|$ we need a family of paths over any two elements of X satisfying a certain coherence condition. For $x, y : X$ we get a path $\mathsf{sip}\ \varphi_{xy} : x \equiv y$. The corresponding coherence condition states that for $x, y, z : X$, we need a path $\mathsf{sip}\ \varphi_{xz} \equiv \mathsf{sip}\ \varphi_{xy} \bullet \mathsf{sip}\ \varphi_{yz}$ (where $\_\bullet\_$ is path composition). By the functoriality of sip, which follows from the functoriality of ua, this path type is equivalent to $\mathsf{sip}\ \varphi_{xz} \equiv \mathsf{sip}\ (\varphi_{yz} \circ \varphi_{xy})$. But by assumption we have $\varphi_{xz} \equiv \varphi_{yz} \circ \varphi_{xy}$, so by applying sip to this path we are done. ◀

For the next lemma, note that for any category $\mathcal{C}$ and family $P : \mathcal{C} \to \mathsf{Type}$, we have the subcategory $\mathcal{C}_{\|P\|}$ of $\mathcal{C}$ induced by $\lambda\ x \to \|\ P(x)\ \| : \mathcal{C} \to \mathsf{hProp}$.

▶ **Lemma 12.** *Let $\mathcal{C}$ be a category with a family $P : \mathcal{C} \to \mathit{Type}$ and a family of R-algebras $\mathcal{F} : \big(\Sigma[\ x \in \mathcal{C}\ ]\ P(x)\big) \to R\text{-}\mathit{Alg}$. Assume furthermore that for $x, y : \mathcal{C}$, $p : P(x)$, $q : P(y)$ with an arrow $f : \mathcal{C}\,[x, y]$ we have*

$$\mathit{isContr}\Big(\mathit{Hom}_R\big[\mathcal{F}(y\ ,\ q)\ ,\ \mathcal{F}(x\ ,\ p)\big]\Big)$$

We can then construct a "universal" presheaf

$$\mathcal{P}_u : \big(\mathcal{C}_{\|P\|}\big)^{op} \to R\text{-}\mathit{Alg}$$

such that for $x : \mathcal{C}$ with $p : P(x)$ we have

$$\mathcal{P}_u(x\ ,\ |\ p\ |) = \mathcal{F}(x\ ,\ p)$$

definitionally, and for $y : \mathcal{C}$, $q : P(y)$ with arrow $f : \mathcal{C}\,[x, y]$, $\mathcal{P}_u\,(f)$ is the unique R-algebra morphism from $\mathcal{F}(y\ ,\ q)$ to $\mathcal{F}(x\ ,\ p)$.

Proof. We first describe the action of $\mathcal{P}_u$ on objects. By currying we fix $x : \mathcal{C}$ and need to provide a function $\|\ P(x)\ \| \to R$-Alg. For this we apply Lemma 11 to $\mathcal{F}(x\ ,\ \_) : P(x) \to R$-$\mathsf{Alg}$. From our contractibility assumption it follows that given $p, q : P(x)$ there are unique morphisms from $\mathcal{F}(x\ ,\ p)$ to $\mathcal{F}(x\ ,\ q)$ and vice versa, so $\mathcal{F}(x\ ,\ p) \cong \mathcal{F}(x\ ,\ q)$. It remains to check that the family of isomorphisms thus defined is closed under composition in the sense of Lemma 11. Again, this follows from contractibility.

For the action of $\mathcal{P}_u$ on morphisms, we start by proving something stronger. Given $x, y : \mathcal{C}$, $p : \|\ P(x)\ \|$, $q : \|\ P(y)\ \|$ with an arrow $f : \mathcal{C}[x, y]$, we have:

$$\mathsf{isContr}\ \Big(\mathrm{Hom}_R\ \big[\ \mathcal{P}_u(x\ ,\ p)\ ,\ \mathcal{P}_u(y\ ,\ q)\ \big]\Big)$$

As being contractible is a proposition, we can assume that $p = | \ p' \ |$ and $q = | \ q' \ |$. In this case $\mathcal{P}_u(x \ , \ p) = \mathcal{F}(y \ , \ p')$ and $\mathcal{P}_u(y \ , \ q) = \mathcal{F}(y \ , \ q')$ and we can just use our contractibility hypothesis. Since a morphism between $(x \ , \ p)$ and $(y \ , \ q)$ in $\mathcal{C}_{\|P\|}$ is just a morphism $f : \mathcal{C}[x, y]$, we can take $\mathcal{P}_u(f)$ to be the center of contraction of the contractible type of R-algebra morphisms above. The functoriality of $\mathcal{P}_u$ then follows immediately. ◀

We now want to apply this construction to the Zariski lattice (seen as a poset category). In the situation of Lemma 12 with $\mathcal{C} = \mathcal{L}_R$ we set, for $\mathfrak{a} : \mathcal{L}_R$:

$$P(\mathfrak{a}) \ = \ \Sigma[\ f \in R \ ] \ (D(f) \equiv \mathfrak{a}) \qquad \text{and} \qquad \mathcal{F}(\mathfrak{a} \ , \ f \ , \ p) \ = \ R[1/f].$$

If we are given $\mathfrak{b} \leq \mathfrak{a}$ with $D(f) \equiv \mathfrak{a}$ and $D(g) \equiv \mathfrak{b}$ then $D(g) \leq D(f)$ and the type of R-algebra morphisms from $R[1/f]$ to $R[1/g]$ is contractible. This way we obtain the desired

$$\mathcal{P}_u : \left(\mathcal{B}_R\right)^{op} \to R\text{-}\mathsf{Alg}$$

Composing with the forgetful functor from R-algebras to commutative rings gives us the desired presheaf on basic opens, denoted by $\mathcal{O}^B$. From this we finally obtain the structure (pre-)sheaf $\mathcal{O} : \left(\mathcal{L}_R\right)^{op} \to \mathsf{CommRing}$ using the right Kan extension machinery described in Section 4. The following fact then becomes rather straightforward to verify:

▶ **Proposition 13.** *For any $f : R$ we get a path $\mathcal{O}\big(D(f)\big) \equiv R[1/f]$.*

Proof. There is a canonical proof $p_f = | \ f \ , \ \mathsf{refl} \ |$ of $D(f)$ belonging to the basic opens. Since we have a natural isomorphism between $\mathcal{O}^B$ and $\mathcal{O} \circ \mathsf{fst}$, we can use the SIP for commutative rings to obtain a path $\mathcal{O}\big(D(f)\big) \equiv \mathcal{O}^B\big(D(f) \ , \ p_f\big)$. But in R-algebras $\mathcal{P}_u\big(D(f) \ , \ p_f\big)$ equals $R[1/f]$ definitionally and applying the forgetful functor to this gives us $R[1/f]$ as a commutative ring (unfortunately not by refl). ◀

As a corollary we obtain the standard sanity check:

▶ **Corollary 14.** $\mathcal{O}\big(\top_{\mathcal{L}_R}\big) \equiv \mathcal{O}\big(D(1)\big) \equiv R.$

Proof. $D(1)$ is the top element of the Zariski lattice by definition, so the first path is just refl. By Proposition 13 we get that $\mathcal{O}\big(D(1)\big) \equiv R[1/1]$. Combining this with Lemma 3.2, we get the desired path. ◀

It remains to prove that $\mathcal{O}^B$ is indeed a sheaf. At this point the standard strategy is to reduce the general case of a cover $D(h) \equiv \bigvee_{i=1}^{n} D(f_i)$ to the special case $h = 1$ and then proceed by some algebraic computations in the rings $R[1/f_i]$.[11] Informally this reduction step follows from a short argument, but it identifies certain localizations by appealing to their canonical isomorphisms. Making this formal in a system without univalence requires to take the isomorphisms at face value and results in cumbersome diagram chases. This problem is described in detail in [4]. There the ultimate breaking point was identifying the rings $R[1/f][1/g]$ and $R[1/fg]$. As the authors point out, simply providing a path between those rings does not solve the problem at hand, since what is actually needed is a path between the diagrams occurring in the sheaf condition. For the remainder of this section we want to show that we can conclude that $\mathcal{O}^B$ is a sheaf from the aforementioned special case, using the observation that the canonical morphisms are unique in R-algebras. In our formalization, the special case of covers of $D(1)$ reads as follows:

[11] See for example [15, theorem 2.33.], [13, theorem 1.3.7] or [18, theorem V.3.3]. Note that in these classical textbooks the sheaf property only has to be verified for finite covers because basic opens are quasi-compact. In contrast, we are restricted to finite covers by definition.

▶ **Lemma 15.** *For a ring A with $f_1, \dots, f_n : A$ such that $1 \in \langle f_1, \dots, f_n \rangle$, we have*

$$A \equiv \varprojlim \left\{ A[{}^1\!/_{f_i}] \to A[{}^1\!/_{f_i f_j}] \leftarrow A[{}^1\!/_{f_j}] \mid 1 \leq i < j \leq n \right\}$$

More precisely, the canonical cone of A over the diagram above is a limiting cone.

Proof. The proof follows closely the textbook approach, see e.g. Mac Lane and Moerdijk [24, p. 125], by some hands-on algebra in the different rings involved. It is precisely at this point that working with concrete implementations of the $A[{}^1\!/_{f_i}]$ as set quotients really simplifies the formalization. ◀

Reducing the sheaf property of $\mathcal{O}^B$ to Lemma 15 can now be done using the special nature of $\mathcal{P}_u$. We also need that the forgetful functor preserves and reflects limits and some basic results about dependent paths. In the library this is packaged up in a generalized, technical lemma, working for arbitrary diagrams, not only those needed for the sheaf property. For the sake of readability however, we proceed to prove our main result directly.

▶ **Theorem 16.** *$\mathcal{O}^B$ is a sheaf on the basic opens.*

Proof. Again for readability, we restrict ourselves to the case of binary covers, i.e. the situation where $D(h) \equiv D(f) \vee D(g)$ for $f, g, h : R$. As described in the proof of Lemma 8, in this case the sheaf property can be reformulated as stating that sq below is a pullback.

$$\begin{tikzcd}
\mathcal{O}^B(D(h)\,,\,p_h) \arrow[r] \arrow[d] \arrow[dr, phantom, "\mathsf{sq}"] & \mathcal{O}^B(D(g)\,,\,p_g) \arrow[d] \\
\mathcal{O}^B(D(f)\,,\,p_f) \arrow[r] & \mathcal{O}^B(D(fg)\,,\,p_{fg})
\end{tikzcd}
\qquad
\begin{tikzcd}
R[{}^1\!/_h] \arrow[r] \arrow[d] \arrow[dr, phantom, "\mathsf{sq}_R"] & R[{}^1\!/_g] \arrow[d] \\
R[{}^1\!/_f] \arrow[r] & R[{}^1\!/_{fg}]
\end{tikzcd}$$

Here the p's are, as in the proof of Proposition 13, the canonical proofs that the D's are in fact basic opens. Note that by definition, sq is obtained by applying the forgetful functor to sq_R and since the forgetful functor *preserves limits* (and in particular pullbacks) it suffices to prove that sq_R is a pullback in R-algebras.

The assumption $D(h) \equiv D(f) \vee D(g)$ gives us $\sqrt{\langle h \rangle} \equiv \sqrt{\langle f, g \rangle}$ and by some standard algebra $1 \in \langle {}^f\!/_1, {}^g\!/_1 \rangle$ in $R[{}^1\!/_h]$. This lets us apply Lemma 15 with $A = R[{}^1\!/_h]$ and we get that sq^* is a pullback (in rings):

$$\begin{tikzcd}
R[{}^1\!/_h] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[dr, phantom, "\mathsf{sq}^*"] & R[{}^1\!/_h][{}^1\!/_g] \arrow[d] \\
R[{}^1\!/_h][{}^1\!/_f] \arrow[r] & R[{}^1\!/_h][{}^1\!/_{fg}]
\end{tikzcd}$$

As all the vertices of sq^* are R-algebras, by the canonical morphisms coming from R, and all the edges of sq^* commute with these canonical morphisms, we can lift sq^* to a square sq^*_R in R-algebras. Since the forgetful functor *reflects limits* (and thus pullbacks), we get that sq^*_R is a pullback square as well.

All that we need is a path $\mathsf{sq}^*_R \equiv \mathsf{sq}_R$ and we are done, as we can transport *the property of being a pullback square* along this path of squares. It is immediate in `Cubical Agda` that to give a path between squares we need to give four paths between the respective vertices and four dependent paths between the morphisms over the paths of vertices. In order to see how this applies to our situation, let us first look at the left side of sq^*_R and sq_R. We get the following square where we have to provide paths at the top and bottom and a dependent path filling this square connecting the vertical arrows ψ and φ:

$$\begin{tikzcd}
R[1/h] \arrow[r, equal] \arrow[d, "\psi"'] & R[1/h] \arrow[d, "\varphi"] \\
R[1/h][1/f] \arrow[r, equal] & R[1/f]
\end{tikzcd}$$

For the top path we just choose refl. For the bottom we apply Lemma 3 and get a path

$$R[1/h][1/f] \equiv R[1/hf] \equiv R[1/f]$$

where the first path is just Lemma 3.1 and the second path is Lemma 3.2 using the fact that $D(hf) \equiv D(f)$ by absorption. Let p denote the composition of these two paths. The dependent path between ψ and φ is then of type

$$\mathsf{PathP}\ \Big(\lambda\ i \to \mathrm{Hom}_R\big[R[1/h]\,,\ p\ i\,\big]\Big)\ \psi\ \varphi$$

By a standard result about PathP, this is equivalent to the non-dependent path type

$$\mathsf{transport}\ \Big(\lambda\ i \to \mathrm{Hom}_R\big[R[1/h]\,,\ p\ i\,\big]\Big)\ \psi \equiv \varphi$$

But by definition φ is the center of contraction of the type $\mathrm{Hom}_R\big[R[1/h]\,,\ R[1/f]\,\big]$. By contractibility, we hence get a path to the transport of ψ and thus the desired dependent path. Repeating this strategy four times, as described in the diagram below, gives us the desired path $\mathsf{sq}^*_R \equiv \mathsf{sq}_R$ and finishes the proof.

$$\begin{tikzcd}
R[1/h] \arrow[rrr, "\exists!"] \arrow[ddd] \arrow[dr, equal] & & & R[1/g] \arrow[ddd] \arrow[dl, equal] \\
& R[1/h] \arrow[r] \arrow[d] \arrow[dr, phantom, "\lrcorner", very near start] \arrow[ll, phantom, "\Longleftarrow \mathsf{PathP} =\!=", near end] \arrow[r, phantom, "\Uparrow \mathsf{PathP} \Vert"', shift left=6] & R[1/h][1/g] \arrow[d] \arrow[rr, phantom, "=\!= \mathsf{PathP} \Longrightarrow", near start] & \\
& R[1/h][1/f] \arrow[r] \arrow[dl, equal] \arrow[r, phantom, "\Vert\ \mathsf{PathP} \Downarrow"', shift right=6] & R[1/h][1/fg] \arrow[dr, equal] & \\
R[1/f] \arrow[rrr, "\exists!"'] & & & R[1/fg]
\end{tikzcd}$$

with the outer vertical arrows labelled $\exists!$.

◀

Combining this with Theorem 10 we get:

▶ **Corollary 17.** *$\mathcal{O}$ is a sheaf on the Zariski lattice $\mathcal{L}_R$.*

Most of the argument in the proof of Theorem 16, including the crucial transport goes through for the general $\mathcal{P}_u$ construction and cones over *arbitrary diagrams*. If we take the action of $\mathcal{P}_u$ on any cone of any shape, we only need two things for establishing that this is a limiting cone: first, a limiting cone in R-algebras of the same shape and second, a family of paths between the corresponding vertices of the two cones. In the case of structure sheaf the limiting cone is provided by Lemma 15 and the paths are provided by Lemma 3. As a matter of fact, the general case is actually easier to formalize and computationally better behaved, even though the pullback case is easier to visualize.

6 Conclusion

In this paper we presented a fully constructive and predicative formalization of the structure sheaf on the Zariski lattice in `Cubical Agda`. To this end, we gave a construction of the Zariski lattice associated to a commutative ring that does not increase the universe level even when working predicatively. We formalized the notion of sheaf on a distributive lattice and formally proved the first steps towards a "comparison lemma" for distributive lattices. In particular, we showed how to extend a sheaf defined on the basis of a lattice, and taking values in any complete category, to a sheaf on the whole lattice. Applying this to the Zariski lattice we then constructed the structure sheaf on its basis. We had to solve higher coherence conditions in order to show that this construction is well-defined. The main insight was that by essentially regarding the structure sheaf to be valued in algebras, not rings, we could use contractibility to solve the coherence issues. Furthermore, it was the same contractibility result that let us formalize the textbook proof of the sheaf property with the help of some univalent machinery.

As discussed in the introduction nothing in the paper crucially relied on cubical features, but they proved convenient in the formalization. In particular, having more things holding by refl, eliminators computing also for higher constructors, and having direct access to dependent paths in the form of PathP types simplified many of the formal proofs. We hope nevertheless that the main ideas introduced in this paper could prove useful for formalizations in other systems. For the remainder of this paper we want to make a few comments that should help putting our work into context.

6.1 Comparison to the classical definition of affine schemes

Even though the constructive, predicative approach described in this paper is similar to the standard, classical textbook approach to affine schemes in the sense that it involves a "lifting" from basic opens, it might not be immediately clear whether we loose anything by working with the Zariski lattice and finitary lattice sheaves. As mentioned in the introduction, from a classical perspective this is not the case because $\mathsf{Spec}\,R$ is a *coherent* space. A topological space X is coherent if it is *compact*, *sober* (its non-empty irreducible closed subsets are the closure of a single point), and its compact opens are closed under finite intersections and form a basis of the topology of X. A coherent map between coherent spaces X and Y is a continuous map $f : X \to Y$ such that for any compact open $K \subseteq Y$, its pre-image $f^{-1}(K)$ is compact as well. Stone's representation theorem for distributive lattices [29] states that the functor from the category of coherent spaces with coherent maps to distributive lattices, sending a coherent space to the lattice of its compact opens, is an equivalence of categories.[12] For the inverse direction we take a distributive lattice and recover the opens of the corresponding space by taking *ideals* on that lattice. We can even recover the points of the space by taking *prime filters* on the lattice. In the case of $\mathsf{Spec}\,R$ the prime filters of $\mathcal{L}_R$ are just the complements of prime ideals of R.[13]

The approach of defining $\mathcal{L}_R$ through formal generators $D(f)$ and obtaining the locale of Zariski opens as the ideals of $\mathcal{L}_R$, is taken in Johnstone's "Stone Spaces" [18, Chap. V.3]. The structure sheaf on the resulting locale of $\mathcal{L}_R$-ideals can then be constructed by only defining it on the base elements $D(f)$. In our predicative and constructive setting we only extend the

[12] Furthermore, any coherent space is coherently homeomorphic to $\mathsf{Spec}\,R$ for some ring R [17], i.e. Spec as a functor from commutative rings to coherent spaces is essentially surjective.

[13] See also the discussion by Coquand, Lombardi and Schuster in the introduction of [9].

structure sheaf construction on basic opens to $\mathcal{L}_R$. Again, classically no information is lost. Whether one considers the structure sheaf to be defined on $\mathsf{Spec}\,R$ as a topological space, on the locale of $\mathcal{L}_R$-ideals or only on $\mathcal{L}_R$, it is determined (up to unique isomorphism) by what happens at the level of basic opens.

More generally, for any coherent space X, the category of sheaves on X is equivalent to the category of (finitary) lattice-sheaves on the compact opens of X. This follows from the comparison lemma for topological spaces, which gives us an equivalence between sheaves on X and sheaves on the basis of compact opens of X. But since the compact opens are all compact we only have to consider finite covers for the sheaf property, which gives us the equivalence to lattice-sheaves on compact opens. Formalizing this classical fact would certainly be interesting in its own right. But as we are interested in the formalization of constructive mathematics, we will just see this fact as a justification that the notion of constructive affine scheme that we arrive at is not fundamentally weaker than the standard classical definition.

6.2 Existing formalizations

To our knowledge, we have presented the first constructive and predicative formalization of affine schemes. However, there are several classical formalizations of affine and general schemes in the literature by now. Examples include an early setoid-based formalization in `Coq` by Chicli [6], the aforementioned formalization in `Lean`'s `mathlib` [4], a more recent formalization in `Isabelle/HOL` [3], and a univalent `Coq` formalization in the `UniMath` library [5]. It is noteworthy that none of these formalizations define the structure sheaf on basic opens first. Instead, they follow the approach of Hartshorne's classic textbook "Algebraic Geometry" [16]. This approach directly defines the structure sheaf on arbitrary opens, but is inherently non-constructive. Assuming classical reasoning (including the axiom of choice) it is quite straightforward to formalize Hartshorne's definition. As a result, the `UniMath` formalization [5] does not actually use univalence in its definition of the structure sheaf.

It should be mentioned however, that in the beginning the `Lean` formalization [4] did use the "lift from basic opens approach". Being unable to formalize the notion of "canonical isomorphism" between localizations $R[1/f]$ in a satisfactory way, `Lean`'s `mathlib` [26] consequently adopted a non-standard take on localizations. Ultimately, the definition of the structure sheaf got completely overhauled using the Hartshorne approach. Buzzard et al. argue in [4, Sect. 3.4] that even with the structure sheaf directly defined using univalence, proving the sheaf property would run into the same problems that they encountered. As the equality/path obtained by an application of the univalence axiom would still carry around the isomorphism in question, it is a priori unclear what has actually been gained by working with paths, as opposed to working with isomorphisms directly. One of the main results of this paper is that on the contrary we can use univalence in a genuinely helpful way to construct the structure sheaf on basic opens and prove its sheaf property. This is achieved by shifting the focus to R-algebras, where the canonical isomorphisms between localizations become the center of contraction of the corresponding path spaces. Indeed, the localizations $R[1/f]$ form a full subcategory of the category of R-algebras that is posetal and equivalent to the poset of basic opens.

6.3 Different univalent approaches to basic opens

One of the main challenges of our formalization was to solve the higher coherence issues when constructing the structure presheaf on basic opens. These coherence issues arose because the basic opens were defined as a subset of the Zariski lattice (i.e. as functions into propositions) using propositional truncation. In constructive mathematics it is common to define subsets X as sets A with an embedding $i : A \hookrightarrow X$ and one can prove in HoTT/UF that these two notions of subsets are equivalent. This raises the question whether one could define the type of basic opens more directly, thus eliminating the coherence issues.

The basic opens can be defined as a quotient on R, equating any f and g such that $\sqrt{\langle f \rangle} = \sqrt{\langle g \rangle}$. A first, now deprecated, formalization attempt defined the structure sheaf on this type. However, in this case we need to map from a set quotient into an groupoid, which is notoriously hard. The general characterization of such maps given by Kraus and von Raumer [21, Thm. 13] is not easily applicable in this case. As a result, we ended up working in R-algebras because the contractibility of the path spaces between localizations solved the coherence issues in this case as well. Rijke has since suggested, in private communications, that the basic opens can be seen as the Rezk completion [33, Sec. 9.9] of R as a poset category with the pre-order $f \leq g$ given by inclusion $\sqrt{\langle f \rangle} \subseteq \sqrt{\langle g \rangle}$. This could potentially be used for an alternative development where coherence issues are avoided altogether.

6.4 Towards constructive quasi-compact, quasi-separated schemes

The structure sheaf, as constructed in this paper, lets us define constructive affine schemes. This is of course only the first step towards a formalization of *constructive schemes.* Schemes are classically defined as a special class of *locally ringed spaces.* However, in the constructive, predicative setting of [10] we are confined to *ringed lattices*, i.e. distributive lattices equipped with a sheaf valued in commutative rings. These correspond to ringed coherent spaces. Maps between those are maps of ringed spaces where the underlying continuous map is coherent. Morphisms of schemes, however, are just morphisms of locally ringed spaces, i.e. morpisms of ringed spaces that induce local morphisms on the stalks. In general these two types of morphisms do not coincide.

Fortunately, the situation is well-behaved for *quasi-compact, quasi-separated* schemes, a very important class of schemes that, in particular, encompasses all *Noetherian* schemes.[14] They are actually just the schemes where the underlying topological space is coherent. Furthermore, if X and Y are quasi-compact, quasi-separated schemes, for any morphism of locally ringed spaces $(f, f^\sharp) : (X, \mathcal{O}_X) \to (Y, \mathcal{O}_Y)$, the underlying continuous map f is coherent. As pointed out in [10], this was essentially already proved by Grothendieck [13, Sec. 6.1]. This makes the constructive lattice-based approach to quasi-compact, quasi-separated schemes as worked out in [10] possible.

Such an approach still needs to be able to talk about morphisms of quasi-compact, quasi-separated schemes, i.e. morphisms of locally ringed spaces. This problem is circumvented in [10] by considering *locally affine morphisms.* A locally affine morphism is induced by ring homomorphisms on affine covers and it is a standard exercise to show that for general schemes this is equivalent to a morphism of locally ringed spaces. For a formalization however, it could be advantageous to work with a constructive reformulation of morphisms of locally ringed spaces. Schuster discusses the right constructive, point-free notion of a morphism of locally ringed spaces in the setting of formal topology in [28]. Transferring this to a development based on ringed lattices could lead to a constructive account of quasi-compact, quasi-separated schemes closer to the usual classical presentation and easier to formalize.

[14] Deligne in fact argued that this class of schemes is actually sufficient for a lot of applications in algebraic geometry [12].

References

1 Carlo Angiuli, Evan Cavallo, Anders Mörtberg, and Max Zeuner. Internalizing representation independence with univalence. *Proc. ACM Program. Lang.*, 5(POPL), January 2021. `doi:10.1145/3434293`.

2 Michael Francis Atiyah and Ian Grant MacDonald. *Introduction to Commutative Algebra.* Addison-Wesley-Longman, 1969.

3 Anthony Bordg, Lawrence Paulson, and Wenda Li. Simple Type Theory is not too Simple: Grothendieck's Schemes Without Dependent Types. *Experimental Mathematics*, 0(0):1–19, 2022. `doi:10.1080/10586458.2022.2062073`.

4 Kevin Buzzard, Chris Hughes, Kenny Lau, Amelia Livingston, Ramon Fernández Mir, and Scott Morrison. Schemes in lean. *Experimental Mathematics*, 0(0):1–9, 2021. `doi:10.1080/10586458.2021.1983489`.

5 Tim Cherganov. Sheaf of rings on Spec R, 2022. URL: `https://github.com/UniMath/UniMath/blob/0df0949b951e198c461e16866107a239c8bc0a1e/UniMath/AlgebraicGeometry/Spec.v`.

6 Laurent Chicli. Une formalisation des faisceaux et des schémas affines en théorie des types avec Coq. Technical Report RR-4216, INRIA, June 2001. URL: `https://hal.inria.fr/inria-00072403`.

7 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. `doi:10.4230/LIPIcs.TYPES.2015.5`.

8 Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS 2018, pages 255–264, New York, NY, USA, 2018. ACM. `doi:10.1145/3209108.3209197`.

9 Thierry Coquand, Henri Lombardi, and Peter Schuster. The projective spectrum as a distributive lattice. *Cahiers de Topologie et Géométrie différentielle catégoriques*, 48(3):220–228, 2007.

10 Thierry Coquand, Henri Lombardi, and Peter Schuster. Spectral schemes as ringed lattices. *Annals of Mathematics and Artificial Intelligence*, 56(3):339–360, 2009.

11 Tom de Jong and Martín Hötzel Escardó. On Small Types in Univalent Foundations. *Logical Methods in Computer Science*, Volume 19, Issue 2, May 2023. `doi:10.46298/lmcs-19(2:8)2023`.

12 Pierre Deligne and Jean-François Boutot. Cohomologie étale: les points de départ. In *Cohomologie Etale*, pages 4–75, Berlin, Heidelberg, 1977. Springer Berlin Heidelberg.

13 Jean Dieudonné and Alexandre Grothendieck. *Éléments de géométrie algébrique*, volume 1. Springer Berlin Heidelberg New York, 1971.

14 Luis Español. Le spectre d'un anneau dans l'algèbre constructive et applications à la dimension. *Cahiers de Topologie et Géométrie Différentielle Catégoriques*, 24(2):133–144, 1983. URL: `http://www.numdam.org/item/CTGDC_1983__24_2_133_0/`.

15 Ulrich Görtz and Torsten Wedhorn. *Algebraic geometry.* Springer, 2010.

16 Robin Hartshorne. *Algebraic geometry*, volume 52. Springer Science & Business Media, 2013.

17 Melvin Hochster. Prime ideal structure in commutative rings. *Transactions of the American Mathematical Society*, 142:43–60, 1969.

18 Peter T. Johnstone. *Stone spaces*, volume 3. Cambridge university press, 1982.

19 André Joyal. Les théoremes de chevalley-tarski et remarques sur l'algèbre constructive. *Cahiers Topologie Géom. Différentielle*, 16:256–258, 1976.

20 Nicolai Kraus. The general universal property of the propositional truncation. In Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau, editors, *20th International Conference on Types for Proofs and Programs (TYPES 2014)*, volume 39 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 111–145, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2014.111.

21 Nicolai Kraus and Jakob von Raumer. Coherence via well-foundedness: Taming set-quotients in homotopy type theory. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '20, pages 662–675, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3373718.3394800.

22 Henri Lombardi and Claude Quitté. *Commutative Algebra: Constructive Methods: Finite Projective Modules*, volume 20. Springer, 2015.

23 Saunders Mac Lane. *Categories for the working mathematician*, volume 5. Springer Science & Business Media, 2013.

24 Saunders Mac Lane and Ieke Moerdijk. *Sheaves in geometry and logic: A first introduction to topos theory*. Springer Science & Business Media, 2012.

25 Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In H. E. Rose and J. C. Shepherdson, editors, *Logic Colloquium '73*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975. doi:10.1016/S0049-237X(08)71945-1.

26 The mathlib Community. The lean mathematical library. In *Proceedings of the 9th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2020, pages 367–381, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3372885.3373824.

27 Ray Mines, Fred Richman, and Wim Ruitenburg. *A course in constructive algebra*. Springer Science & Business Media, 2012.

28 Peter Schuster. The zariski spectrum as a formal geometry. *Theoretical Computer Science*, 405(1):101–115, 2008. Computational Structures for Modelling Space, Time and Causality. doi:10.1016/j.tcs.2008.06.030.

29 Marshall Harvey Stone. Topological representations of distributive lattices and brouwerian logics. *Časopis pro pěstování matematiky a fysiky*, 67(1):1–25, 1938.

30 Thomas Streicher. *Investigations Into Intensional Type Theory*. Habilitation thesis, Ludwig-Maximilians-Universität München, 1993. URL: https://www2.mathematik.tu-darmstadt.de/~streicher/HabilStreicher.pdf.

31 The Agda Development Team. The Agda programming language. URL: http://wiki.portal.chalmers.se/agda/pmwiki.php.

32 Ayberk Tosun and Martín Hötzel Escardó. Patch locale of a spectral locale in univalent type theory, 2023. arXiv:2301.04728.

33 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. https://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

34 Ravi Vakil. The rising sea: Foundations of algebraic geometry, 2017. draft. URL: https://math.stanford.edu/~vakil/216blog/FOAGnov1817public.pdf.

35 Andrea Vezzosi, Anders Mörtberg, and Andreas Abel. Cubical agda: A dependently typed programming language with univalence and higher inductive types. *Journal of Functional Programming*, 31:e8, 2021. doi:10.1017/S0956796821000034.

36 V. Voevodsky, B. Ahrens, D. Grayson, et al. UniMath: Univalent Mathematics. Available at https://github.com/UniMath.

37 Vladimir Voevodsky. Univalent foundations, September 2010. Notes from a talk in Bonn. URL: https://www.math.ias.edu/vladimir/sites/math.ias.edu.vladimir/files/Bonn_talk.pdf.

38 Vladimir Voevodsky. Resizing rules – their use and semantic justification. slides from a talk at types, bergen, 11 september, 2011.

39 Vladimir Voevodsky. An experimental library of formalized mathematics based on the univalent foundations. *Mathematical Structures in Computer Science*, 25(5):1278–1294, 2015. doi:10.1017/S0960129514000577.

Univalent Monoidal Categories

Kobe Wullaert

Delft University of Technology, The Netherlands

Ralph Matthes

IRIT, Université de Toulouse, CNRS, Toulouse INP, UT3, France

Benedikt Ahrens

Delft University of Technology, The Netherlands
University of Birmingham, UK

Abstract

Univalent categories constitute a well-behaved and useful notion of category in univalent foundations. The notion of univalence has subsequently been generalized to bicategories and other structures in (higher) category theory. Here, we zoom in on monoidal categories and study them in a univalent setting. Specifically, we show that the bicategory of univalent monoidal categories is univalent. Furthermore, we construct a Rezk completion for monoidal categories: we show how any monoidal category is weakly equivalent to a univalent monoidal category, universally. We have fully formalized these results in UniMath, a library of univalent mathematics in the Coq proof assistant.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Logic and verification

Keywords and phrases Univalence, Monoidal categories, Rezk completion, Displayed (bi)categories, Proof assistant Coq, UniMath library

Digital Object Identifier 10.4230/LIPIcs.TYPES.2022.15

Funding *Benedikt Ahrens*: This work was partially funded by EPSRC under agreement number EP/T000252/1.

Acknowledgements We gratefully acknowledge the work by the Coq development team in providing the Coq proof assistant and surrounding infrastructure, as well as their support in keeping UniMath compatible with Coq. Furthermore, we thank Niels van der Weide for helpful discussions on the subject matter and for reviewing the formalization. We also thank Vikraman Choudhury for a question regarding the connection of the monoidal Rezk completion to the Day convolution product. Furthermore, we thank the anonymous referees for their helpful feedback; besides triggering further thoughts on the technical aspects, it helped us to improve the presentation.

1 Introduction

When working in univalent foundations (see [15]), definitions have to be designed carefully in order to correspond, via the intended semantics, to the *expected* notions in set-theoretic foundations. The notion of univalent category [2] has been shown to be a good notion, in the sense that it corresponds to the usual notion of category under Voevodsky's model in simplicial sets [9].[1] Examples of univalent categories are plentiful, but not all categories arising in practice – for instance when studying categorical semantics of type theory – are univalent. In [2], the authors give a construction of a "free" univalent category from any category $\mathcal{C}$, which they call the Rezk completion of $\mathcal{C}$.

Since then, the univalence condition and completion operation have been studied further.

[1] To emphasize that univalent categories are the right notion of category in univalent foundations, they are just called "categories" in [2].

28th International Conference on Types for Proofs and Programs (TYPES 2022).
Editors: Delia Kesner and Pierre-Marie Pédrot; Article No. 15; pp. 15:1–15:21
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Firstly, in [16], Van der Weide constructs a class of higher inductive types using the groupoid quotient. It is shown that the groupoid quotient gives rise to a biadjunction between the bicategory of groupoids and the bicategory of 1-types (which is isomorphic to the bicategory of univalent groupoids); the left adjoint thus yields a univalent completion operation for groupoids. Van der Weide furthermore lifts this completion to "structured groupoids", that is, to groupoids equipped with an algebra structure for some endo-pseudofunctor on (univalent) groupoids.

Secondly, the univalence condition on categories was extended to bicategories in [1] and to other (higher-)categorical structures in [4]. In more detail, [4] develops a notion of theory for mathematical structures, and a notion of univalence for models of such theories.

Thirdly, univalent displayed graphs are used in [5] to define and study higher groups.

In the present paper, we continue the study of univalent (higher-)categorical structures, focusing on *monoidal* categories. Monoidal categories are very useful in a variety of contexts, such as quantum mechanics [7] and computing [6], modeling concurrency [11], probability theory [13] and probabilistic programming [12], and neural networks [10]. We present two results on monoidal categories:

1. We show that the bicategory of univalent monoidal categories is univalent. Here, a univalent monoidal category is a univalent category with a monoidal structure.
2. We construct, for any monoidal category, a monoidal Rezk completion. It is, in particular, a univalent monoidal category; the challenge lies in establishing the universal property of a Rezk completion, here modified for monoidal categories.

Both results have been formalized in the `UniMath` library of univalent mathematics, based on the Coq proof assistant.

The first of these results may be considered to be a basic sanity check; failing to prove this would question the validity of our definitions. However, its proof is technically difficult, and, in our experience, only feasible through the disciplined application of "displayed" technology as developed in [3] and [1].

The second result consists, more specifically, of a lifting of the Rezk completion for categories as constructed in [2] to the monoidal structure. As such, it also relies on displayed technology: the equivalence expressing the universal property of our monoidal Rezk completion is given as a displayed equivalence on top of the equivalence constructed in [2].

Our work is strongly related to some of the work mentioned above.

Firstly, an instance of Van der Weide's work covers monoidal groupoids; see [16, Section 6.7.4]. Compared to that work, our work discusses monoidal *categories* rather than groupoids, but does not cover general structures. In particular, we also provide a completion operation for *lax* and *oplax* monoidal categories. Work on the "pushout" of our and Van der Weide's work, a Rezk completion for structured categories, is ongoing (see also Section 5).

Secondly, [4, Example 8.7] studies monoidal categories. It is shown there that the general univalence condition on a model of the theory of monoidal categories defined in that work simplifies, in the case of monoidal categories, to the underlying category being univalent. Thus, the univalent monoidal categories of [4, Example 8.7] are the same as the ones studied in the present work.

In the remainder of the introduction, we review the Rezk completion and displayed (bi)categories, respectively. We also give some details about the formalization.

▶ **Notation 1.** *In order to stay consistent with the notation used in `UniMath`, we write the composition in diagrammatic order, i. e., the composition of $f : x \to y$ and $g : y \to z$ is denoted as $f \cdot g : x \to z$.*

There are different notions of *sameness* between categories:

▶ **Definition 2.** *A functor $F : \mathcal{C} \to \mathcal{D}$ is called*
1. *a* ***weak equivalence*** *if it is fully faithful and essentially surjective;*
2. *a* ***(strong) equivalence*** *if it is fully faithful and split essentially surjective. Equivalently, this means that F is invertible up to a natural isomorphism;*
3. *an* ***adjoint equivalence*** *is a (strong) equivalence F whose inverse (up to a natural isomorphism) is the right adjoint of F;*
4. *an* ***isomorphism*** *if it is fully faithful and the function on objects is an equivalence of types.*

Even though these four concepts are closely related, they enjoy different properties. The Rezk completion is, in general, only a weak equivalence; categorical structure does not necessarily transfer along a weak equivalence. For strict categories (i. e., categories whose type of objects is a set), the statement that every weak equivalence is an (adjoint) equivalence is equivalent to the axiom of choice. However, if one restricts to univalent categories, these four notions are always equivalent (without using the axiom of choice).

1.1 Review of the Rezk completion for categories

The Rezk completion for categories was constructed in [2]. In essence, given a category $\mathcal{C}$, its Rezk completion is given by a univalent category $\mathsf{RC}(\mathcal{C})$ and a weak equivalence $\mathcal{H} : \mathcal{C} \to \mathsf{RC}(\mathcal{C})$. This weak equivalence has the following property: any functor $F : \mathcal{C} \to \mathcal{E}$, with $\mathcal{E}$ a univalent category, factors *uniquely* via $\mathcal{H}$, as depicted in the following diagram.

$$\begin{tikzcd}
\mathcal{C} \arrow[d, "\mathcal{H}"'] \arrow[dr, "F"] & \\
\mathsf{RC}(\mathcal{C}) \arrow[r, dashed, "\exists!"'] & \mathcal{E}
\end{tikzcd} \tag{1}$$

▶ Remark 3. The universal property satisfied by the Rezk completion is a bicategorical one, see Definition 5. From a purely category-theoretic viewpoint, the factorization in Equation (1) is unique up to natural isomorphism. However, since $\mathcal{E}$ is univalent, the functor category $[\mathsf{RC}(\mathcal{C}), \mathcal{E}]$ is also univalent. Therefore, the factorization of such a functor is unique.

In [2], it is said that the construction gives a universal way to replace a category by a univalent category. This construction is indeed universal in a bicategorical sense, according to the following lemma:

▶ **Lemma 4** ([2, Thm. 8.4], `precomp_adjoint_equivalence`)**.** *Let $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ be a weak equivalence between categories. For any univalent category $\mathcal{E}$, the functor $\mathcal{H} \cdot (-) : [\mathcal{D}, \mathcal{E}] \to [\mathcal{C}, \mathcal{E}]$ is an adjoint equivalence of categories.*

Lemma 4, when applied to the Rezk completion, provides an instance of a "(left) universal arrow":

▶ **Definition 5** (`left_universal_arrow`)**.** *Let $R : \mathcal{B}_2 \to \mathcal{B}_1$ be a pseudo-functor. A* ***left universal arrow*** *from an object $x : (\mathcal{B}_1)_0$ to R is given by:*
1. *an object $L\,x : (\mathcal{B}_2)_0$,*
2. *a morphism $\eta_x : \mathcal{B}_1(x, R(L\,x))$;*
3. *for any $y : (\mathcal{B}_2)_0$, the functor*

$$\eta_x \cdot (R\,-) : \mathcal{B}_2(L\,x, y) \to \mathcal{B}_1(x, R\,y) \enspace ,$$

which acts on morphisms by applying R and whiskering with η_x, is an adjoint equivalence of categories.

▶ Remark 6. Writing **Cat** for the bicategory of categories, functors, and natural transformations, and $\mathbf{Cat}_{univ}$ for the full sub-bicategory of **Cat** consisting of *univalent* categories, functors, and natural transformations, Lemma 4 applied to the Rezk completion of $\mathcal{C}$ provides a universal arrow from $\mathcal{C}$ to the inclusion $\mathbf{Cat}_{univ} \hookrightarrow \mathbf{Cat}$. We expect the following to hold: if we have, for any object x, a left universal arrow to R with object part $L\,x$, then the assignment $x \mapsto L\,x$ induces a pseudo-functor $L : \mathcal{B}_1 \to \mathcal{B}_2$ which is a left biadjoint to R. Hence, Lemma 4 applied to the Rezk completion would yield a left bi-adjoint to the inclusion $\mathbf{Cat}_{univ} \hookrightarrow \mathbf{Cat}$. However, we have not found a reference for the connection between universal arrows and biadjunctions. As we do not need this correspondence, we do not develop it further.

▶ Remark 7. In [2], the Rezk completion has been constructed as the co-restriction of the Yoneda embedding to its image. It is already known how the Yoneda embedding transports the monoidal structure; more details on the connection between these approaches are given in Section 4.5. However, this construction raises the universe level of the type of objects and morphisms. In `https://1lab.dev/Cat.Univalent.Rezk.html`, the authors show how to decrease the universe level of the type of objects by one, using the construction of small images (and, in particular, higher inductive types). One can also construct (the type of objects of) the Rezk completion as a higher inductive type. This has been done in [15].

In this paper, we work with an *abstract* Rezk completion of a category instead of a concrete implementation. Consequently, the approach presented here can be applied to any of those constructions.

1.2 Review of displayed (bi)categories

In this section, we recall the basic concepts of displayed bicategories and their univalence. More information can be found in [1].

Let us first briefly recall the idea of displayed categories.

Many concrete examples of categories are given by structured sets and structure-preserving functions. An example of this is the category **Mon** of monoids and monoid homomorphisms. In particular, an identity morphism is an identity function (i. e., the identity morphism in **Set**) and the composition of monoid homomorphisms is given by the composition of the underlying functions (i. e., the composition in **Set**). Therefore, working in a category of structured sets often means lifting structure of the category **Set** to the additional structure. An example of this phenomenon is the product of monoids: the underlying set of a product of monoids can be constructed as the product of the underlying sets (Example 8).

The notion of **displayed category** formalizes the process of creating a new category out of an old category by adding structure and/or properties on the objects and/or morphisms in the following way: a displayed category ([3, Def. 3.1]) specifies precisely the extra structure and the extra laws needed to build the new category out of the old one. This new category is then called the *total category* of the displayed category ([3, Def. 3.2]).

▶ **Example 8.** The category **Mon** of monoids can be constructed as a total category over **Set** as follows:

1. For X : **Set**, the type of displayed objects over X is the type of monoid structures on X:

$$\sum_{m:X\times X\to X}\sum_{e:X}\mathsf{isAssociative}(m)\times\prod_{x:X}(e\cdot x = x\times x\cdot e = x)\quad,$$

where $\mathsf{isAssociative}(m)$ is the proposition stating that m is associative.

2. Assume given $X, Y : \mathbf{Set}, f : \mathbf{Set}(X, Y)$ and (m_X, e_X, p_X) (resp. (m_Y, e_Y, p_Y)) a displayed object over X (resp. Y), i. e., the structure of a monoid. The type of displayed morphisms over f is the proposition stating that f is a monoid homomorphism from (m_X, e_X, p_X) to (m_Y, e_Y, p_Y):

$$(f\, e_X = e_Y) \times \prod_{x_1, x_2 : X} f\,(m_X(x_1, x_2)) = m_Y(f\, x_1, f\, x_2) \ .$$

Analogously, there is also the notion of a **displayed bicategory**:

▶ **Definition 9** ([1, Def. 6.1], `disp_bicat`). *Let $\mathcal{B}$ be a bicategory. A* ***displayed bicategory*** *$\mathcal{D}$ over $\mathcal{B}$ consists of:*

1. *for any $x : \mathcal{B}$, a type $\mathcal{D}_x$ of* displayed objects *over x,*
2. *for any $f : \mathcal{B}(x, y)$ and $\bar{x} : \mathcal{D}_x$ and $\bar{y} : \mathcal{D}_y$, a type $\mathcal{D}_f(\bar{x}, \bar{y})$ of* displayed morphisms *over f,*
3. *for any $\alpha : \mathcal{B}(x, y)(f, g)$ and $\bar{f} : \mathcal{D}_f(\bar{x}, \bar{y})$ and $\bar{g} : \mathcal{D}_g(\bar{x}, \bar{y})$, a set $\bar{f} \overset{\alpha}{\Longrightarrow} \bar{g}$ of* displayed 2-cells *over α;*

together with a composition of displayed morphisms and displayed 2-cells (over the composition in $\mathcal{B}$) and a displayed identity morphism and 2-cell (over the identity morphism resp. 2-cell in $\mathcal{B}$). The axioms of a bicategory have corresponding displayed axioms (over those axioms in $\mathcal{B}$).

▶ **Definition 10** ([1, Def. 6.2], `total_bicat`). *Let $\mathcal{D}$ be a displayed bicategory over $\mathcal{B}$. The* ***total bicategory*** *of $\mathcal{D}$, denoted as $\int \mathcal{D}$, has as i-cells (with $i = 0, 1, 2$), pairs $(x, \bar{x})$ where x is an i-cell of $\mathcal{B}$ and $\bar{x}$ is a displayed i-cell of $\mathcal{D}$ over x.*

▶ **Example 11.** The bicategory whose objects are categories equipped with a terminal object, whose morphisms are functors preserving the terminal objects (strongly) and whose 2-cells are natural transformations, can be constructed as a total bicategory over **Cat** as follows:

1. For $\mathcal{C} : \mathbf{Cat}$, the type of displayed objects over **Cat** is the type expressing that $\mathcal{C}$ has a terminal object:

$$\sum_{X : \mathcal{C}} \mathsf{isTerminal}(X) \ .$$

2. Assume given $\mathcal{C}, \mathcal{D} : \mathbf{Cat}, F : \mathbf{Cat}(\mathcal{C}, \mathcal{D})$ and $(T_\mathcal{C}, p_\mathcal{C})$ (resp. $(T_\mathcal{D}, p_\mathcal{D})$) displayed objects over $\mathcal{C}$ (resp. $\mathcal{D}$). The type of displayed morphisms over F is the proposition stating that F preserves the terminal object:

$$\mathsf{isIsomorphism}(!) \ ,$$

where ! is the unique morphism $F\, T_\mathcal{C} \to T_\mathcal{D}$ given by the universal property of the terminal object $T_\mathcal{D}$.

3. Let $F, G : \mathbf{Cat}(\mathcal{C}, \mathcal{D})$ be functors between categories $\mathcal{C}$ and $\mathcal{D}$ and assume:
   a. $(T_\mathcal{C}, p_\mathcal{C})$ (resp. $(T_\mathcal{D}, p_\mathcal{D})$) a witness that $\mathcal{C}$ (resp. $\mathcal{D}$) has a terminal object, i. e., it is a displayed object over $\mathcal{C}$ (resp. $\mathcal{D}$),
   b. μ^F (resp. μ^G) a proof witnessing that F (resp. G) preserves the terminal object strongly, i. e., μ^F (resp. μ^G) is a displayed morphism over F (resp. G).

   For any natural transformation $\alpha : F \Rightarrow G$, the type of displayed 2-cells over α is the unit type.

Given displayed bicategories $\mathcal{D}_1$ and $\mathcal{D}_2$ over a bicategory $\mathcal{B}$, we construct the product $\mathcal{D}_1 \times \mathcal{D}_2$ over $\mathcal{B}$. The displayed objects, morphisms, and 2-cells are pairs of objects, morphisms, and 2-cells, respectively (`disp_dirprod_bicat`).

A displayed bicategory is *locally univalent* if the function of type

$$\bar{f} =_p \bar{g} \to \bar{f} \cong_{\mathsf{idtoiso}^{2,1}_{f,g}(p)} \bar{g} \ ,$$

sending refl to the identity displayed isomorphism, is an equivalence of types for all morphisms f and g of the same type, $p : f = g$ and $\bar{f}$ (resp. $\bar{g}$) displayed morphisms over f (resp. g).

A displayed bicategory is *globally univalent* if the function of type

$$\bar{x} =_p \bar{y} \to \bar{x} \simeq_{\mathsf{idtoiso}^{2,0}_{x,y}(p)} \bar{y} \ ,$$

sending refl to the identity displayed adjoint equivalence, is an equivalence of types for all objects x and y, $p : x = y$ and $\bar{x}$ (resp. $\bar{y}$) displayed objects over x (resp. y).

A displayed bicategory is *univalent* if it is both locally and globally univalent (`disp_univalent_2`, `disp_univalent_2_1`, `disp_univalent_2_0`).

▶ **Lemma 12** ([1, Thm. 7.4], `total_is_univalent_2`). *Let $\mathcal{D}$ be a displayed bicategory over $\mathcal{B}$ and $q \in \{locally, globally\}$. Then $\int D$ is q-univalent if $\mathcal{B}$ is q-univalent and $\mathcal{D}$ is q-univalent.*

▶ Remark 13. As witnessed by Lemma 12, certain properties of the total bicategory can be expressed in terms of the *base* bicategory and the displayed bicategory. This allows one to divide a problem, in this case showing univalence, into multiple steps.

Therefore, while we are interested in studying the total bicategory, we usually only describe the displayed bicategory.

▶ **Definition 14** ([1, Defs. 7.7, 7.8], `disp_locally_groupoid`, `disp_2cells_isaprop`). *A displayed bicategory $\mathcal{D}$ over a bicategory $\mathcal{B}$ is called*

1. ***Locally groupoidal** if all displayed 2-cells over invertible 2-cells are invertible;*
2. ***Locally propositional** if each type of displayed 2-cells is a proposition.*

We will also need the displayed analogue of the concept of a functor being essentially surjective:

▶ **Definition 15** (`disp_functor_disp_ess_split_surj`). *A displayed functor $\bar{F} : \mathcal{D}_1 \to \mathcal{D}_2$ over a functor $F : \mathcal{C}_1 \to \mathcal{C}_2$ is **displayed split essentially surjective** if for any $x : \mathcal{C}$ and $\bar{y} : (\mathcal{D}_2)_{F\,x}$, a displayed object $\bar{x} : (\mathcal{D}_1)_x$ is given together with a displayed isomorphism between $\bar{F}\,\bar{x}$ and $\bar{y}$ over the identity isomorphism $\mathsf{Id}_{F\,x}$.*

1.3 Formalization in UniMath

The results presented here are formulated inside intensional dependent type theory. We carefully distinguish between data and properties, i. e., data is always explicitly given which avoids the use of the axiom of choice and the law of excluded middle. The results presented here are formalized and checked in the library `UniMath` [17] of univalent mathematics, based on the proof assistant `Coq` [14].

The formalization referred to in this paper is presented in the `UniMath` commit `6d2d288` (more precisely, the given link leads to the source code repository right after merging this commit). A generated HTML documentation of the sources at this commit is hosted online. Most of our definitions, lemmas, and theorems are accompanied by a link which leads to the corresponding definition, lemma, and theorem in the documentation.

The formalization is built upon the existing library of (bi)category theory and the theory of displayed (bi)categories. The (1-)categorical formulation of displayed categories has been developed in [3] and the bicategorical formulation has been developed in [1].

$$\begin{tikzcd}
 & \mathbf{Cat}_S \arrow[d, hook, "Def.23"] & \\
 & \mathbf{Cat}_P \arrow[d, hook, "Def.22"] & \\
 & \mathbf{Cat}_{UA} \arrow[dl, dashed] \arrow[d, dashed] \arrow[dr, dashed] & \\
\mathbf{Cat}_{LU} \arrow[dr, "Def.19"'] & \mathbf{Cat}_{RU} \arrow[d, "Def.20"'] & \mathbf{Cat}_A \arrow[dl, "Def.21"] \\
 & \mathbf{Cat}_{TU} \arrow[dl, dashed] \arrow[dr, dashed] & \\
\mathbf{Cat}_U \arrow[r, "Def.18"'] & \mathbf{Cat} & \mathbf{Cat}_T \arrow[l, "Def.17"]
\end{tikzcd} \tag{2}$$

Figure 1 Overview of construction steps towards **MonCat** and $\mathbf{MonCat}^{stg}$.

The accompanying code, specific to this work, consists of approximately 7000 lines of code. However, the formalisation also made it necessary to contribute to the `UniMath` library on monoidal categories more generally.

2 The bicategory of monoidal categories

In this section we construct the bicategory **MonCat** (resp. $\mathbf{MonCat}^{stg}$) of monoidal categories, lax (resp. strong) monoidal functors and monoidal natural transformations. We construct this bicategory as the total bicategory of a displayed bicategory over the bicategory **Cat** of categories, functors, and natural transformations.

This displayed bicategory in itself is constructed by stacking different displayed bicategories. First, we construct a displayed bicategory $\mathbf{Cat}_T$ (resp. $\mathbf{Cat}_U$) over **Cat** that adds a tensor (resp. a unit). Then, we construct displayed bicategories $\mathbf{Cat}_{LU}, \mathbf{Cat}_{RU}$ and $\mathbf{Cat}_A$ over the total bicategory of $\mathbf{Cat}_{TU} := \mathbf{Cat}_T \times \mathbf{Cat}_U$ that add the left unitor, right unitor and the associator, respectively. The product of these displayed bicategories is denoted by $\mathbf{Cat}_{UA}$ and the laws that relate the unitors and the associator, e. g., the triangle and pentagon identities, are represented by a full (displayed) sub-bicategory $\mathbf{Cat}_P$ of $\mathbf{Cat}_{UA}$. Lastly, we also have a displayed (sub)bicategory $\mathbf{Cat}_S$ of $\mathbf{Cat}_P$ that enforces the strongness of the monoidal functors.

The construction is summarized in Figure 1. The precise meaning of this diagram is explained in the rest of this section and further explained in Remark 24.

▶ Remark 16. Although the construction of **MonCat** (resp. $\mathbf{MonCat}^{stg}$) is standard (when working in univalent foundations), we explain the construction in quite some detail because both Section 3 and Section 4 heavily depend on the construction of monoidal categories (resp. lax/strong monoidal functors and natural transformations) in this displayed way. In particular, this allows us to fix notation and allows for the big picture of the constructions to become more visible.

The first displayed bicategory we construct adds the structure of a tensor and a unit. Since the unit and tensor are (without the unitors) independent of each other, we can define this as the product of displayed bicategories, the first representing the tensor and the second representing the unit.

▶ **Definition 17** (`bidisp_tensor_disp_bicat`). *The displayed bicategory* $\mathbf{Cat}_T$ *over* $\mathbf{Cat}$ *is defined as follows:*

1. *The displayed objects over a category* $\mathcal{C} : \mathbf{Cat}$ *are the functors of type* $\mathcal{C} \times \mathcal{C} \to \mathcal{C}$*, called* tensors *over* $\mathcal{C}$ *and are denoted by* $\otimes_{\mathcal{C}}$*.*
2. *The displayed morphisms over a functor* $F : \mathcal{C} \to \mathcal{D}$ *from* $\otimes_{\mathcal{C}}$ *to* $\otimes_D$ *are the natural transformations of type* $(F \times F) \cdot \otimes_{\mathcal{D}} \Rightarrow \otimes_{\mathcal{C}} \cdot F$*, called* witnesses of tensor-preservation of F *and are denoted by* μ^F*.*
3. *The displayed* 2*-cells over a natural transformation* $\alpha : F \Rightarrow G$ *from* μ^F *to* μ^G *are the proofs of the proposition*

$$\prod_{x,y:\mathcal{C}} (\alpha_x \otimes_D \alpha_y) \cdot \mu^G_{x,y} = \mu^F_{x,y} \cdot \alpha_{x \otimes_C y} \quad .$$

▶ **Definition 18** (`bidisp_unit_disp_bicat`). *The displayed bicategory* $\mathbf{Cat}_U$ *over* $\mathbf{Cat}$ *is defined such that:*

1. *The displayed objects over a category* $\mathcal{C} : \mathbf{Cat}$ *are the objects of* $\mathcal{C}$*, called* units *over* $\mathcal{C}$ *and are denoted by* $I_{\mathcal{C}}$*.*
2. *The displayed morphisms over a functor* $F : \mathcal{C} \to \mathcal{D}$ *from* $I_{\mathcal{C}}$ *to* $I_{\mathcal{D}}$ *are the morphisms of type* $\mathcal{D}(I_{\mathcal{D}}, FI_{\mathcal{C}})$*, called* witnesses of unit-preservation of F *and are denoted by* ϵ^F*.*
3. *The displayed* 2*-cells over a natural transformation* $\alpha : F \Rightarrow G$ *from* ϵ^F *to* ϵ^G *are the proofs of the proposition*

$$\epsilon^F \cdot \alpha_{I_{\mathcal{C}}} = \epsilon^G \quad .$$

We denote by $\mathbf{Cat}_{TU}$ the displayed bicategory which is the product of $\mathbf{Cat}_T$ and $\mathbf{Cat}_U$ (`bidisp_tensor_unit`).

To fix some notation: The total bicategory $\int \mathbf{Cat}_{TU}$ has as objects triples $(\mathcal{C}, \otimes_{\mathcal{C}}, I_{\mathcal{C}})$ where $\mathcal{C}$ is a category, $\otimes_{\mathcal{C}}$ a tensor on $\mathcal{C}$ and $I_{\mathcal{C}}$ a unit on $\mathcal{C}$. A morphism from $(\mathcal{C}, \otimes_{\mathcal{C}}, I_{\mathcal{C}})$ to $(\mathcal{D}, \otimes_{\mathcal{D}}, I_{\mathcal{D}})$ is a triple (F, μ^F, ϵ^F) where F is a functor of type $\mathcal{C} \to \mathcal{D}$, μ^F a witness of tensor-preservation of F and ϵ^F a witness of unit-preservation of F.

We now add the unitors and the associator. Since they are independent of each other (before adding the triangle and pentagon equalities), we can again define them as a product of displayed bicategories. These displayed bicategories have trivial displayed 2-cells since monoidal natural transformations only use the data of the tensor and the unit. Thus we define these displayed bicategories as displayed categories. The formal construction of turning a displayed category into a displayed bicategory with trivial 2-cells is formalized as `disp_cell_unit_bicat`.

▶ **Definition 19** (`bidisp_lu_disp_bicat`). *The displayed bicategory* $\mathbf{Cat}_{LU}$ *over* $\int \mathbf{Cat}_{TU}$ *is defined as the displayed category (with trivial* 2*-cells) such that:*

1. *The displayed objects over a triple* $(\mathcal{C}, \otimes_C, I_C)$ *are the natural isomorphisms of type* $(I_{\mathcal{C}} \otimes_{\mathcal{C}} -) \Rightarrow \mathsf{Id}_{\mathcal{C}}$*, called* left unitors *over* $(\mathcal{C}, \otimes_C, I_C)$ *and are denoted by* $\lambda^{\mathcal{C}}$*.*
2. *The displayed morphisms over a triple* (F, μ^F, ϵ^F) *from* $\lambda^{\mathcal{C}}$ *to* $\lambda^{\mathcal{D}}$ *are proofs of the proposition:*

$$\prod_{x:\mathcal{C}} (\epsilon^F \otimes_{\mathcal{D}} \mathsf{Id}_{Fx}) \cdot \mu^F_{I_{\mathcal{C}},x} \cdot F\lambda^{\mathcal{C}}_x = \lambda^{\mathcal{D}}_{Fx} \quad .$$

▶ **Definition 20** (`bidisp_ru_disp_bicat`). *The displayed bicategory* $\mathbf{Cat}_{RU}$ *over* $\int \mathbf{Cat}_{TU}$ *is defined as the displayed category (with trivial* 2*-cells) such that:*

1. *The displayed objects over a triple* $(\mathcal{C}, \otimes_C, I_C)$ *are the natural isomorphisms of type* $(- \otimes_{\mathcal{C}} I_{\mathcal{C}}) \Rightarrow \mathsf{Id}_{\mathcal{C}}$*, called* right unitors *over* $(\mathcal{C}, \otimes_C, I_C)$ *and are denoted as* $\rho^{\mathcal{C}}$*.*

2. *The displayed morphisms over a triple* (F, μ^F, ϵ^F) *from* $\rho^{\mathcal{C}}$ *to* $\rho^{\mathcal{D}}$ *are proofs of the proposition:*

$$\prod_{x:\mathcal{C}} (\mathsf{Id}_{Fx} \otimes_{\mathcal{D}} \epsilon^F) \cdot \mu^F_{x,I_{\mathcal{C}}} \cdot F\rho^{\mathcal{C}}_x = \rho^{\mathcal{D}}_{Fx} \quad .$$

▶ **Definition 21** (`bidisp_associator_disp_bicat`)**.** *The displayed bicategory* $\mathbf{Cat}_A$ *over* $\int \mathbf{Cat}_{TU}$ *is defined as the displayed category (with trivial* 2*-cells) such that:*

1. *The displayed objects over a triple* $(\mathcal{C}, \otimes_C, I_C)$ *are the natural isomorphisms of type* $((- \otimes_{\mathcal{C}} -) \otimes_{\mathcal{C}} -) \Rightarrow (- \otimes_{\mathcal{C}} (- \otimes_{\mathcal{C}} -))$*, called* associators *over* $(\mathcal{C}, \otimes_C, I_C)$ *and are denoted as* $\alpha^{\mathcal{C}}$*.*
2. *The displayed morphisms over a triple* (F, μ^F, ϵ^F) *from* $\alpha^{\mathcal{C}}$ *to* $\alpha^{\mathcal{D}}$ *are proofs of the proposition:*

$$\prod_{x,y,z:\mathcal{C}} (\mu^F_{x,y} \otimes_{\mathcal{D}} \mathsf{Id}_{Fz}) \cdot \mu^F_{x \otimes_{\mathcal{C}} y, z} \cdot F\alpha^{\mathcal{C}}_{x,y,z} = \alpha^{\mathcal{D}}_{Fx,Fy,Fz} \cdot (\mathsf{Id}_{Fx} \otimes_{\mathcal{D}} \mu^F_{y,z}) \cdot \mu^F_{x, y \otimes_{\mathcal{C}} z} \quad .$$

We denote by $\mathbf{Cat}_{UA}$ the displayed bicategory over $\int \mathbf{Cat}_{TU}$ which is the product of $\mathbf{Cat}_{LU}, \mathbf{Cat}_{RU}$ and $\mathbf{Cat}_A$ (`bidisp_assunitors_disp_bicat`).

▶ **Definition 22** (`disp_bicat_univmon`)**.** *The displayed bicategory* $\mathbf{Cat}_P$ *is the full displayed sub-bicategory of* $\mathbf{Cat}_{UA}$ *specified by the product of the following predicates:*

1. *Triangle equality:*

$$\prod_{x,y:\mathcal{C}} \alpha_{x,I,y} \cdot \mathsf{Id}_x \otimes \lambda_y = \rho_x \otimes \mathsf{Id}_y \quad .$$

2. *Pentagon equality:*

$$\prod_{w,x,y,z:\mathcal{C}} (\alpha_{w,x,y} \otimes \mathsf{Id}_z) \cdot \alpha_{w,x \otimes y,z} \cdot \mathsf{Id}_w \otimes \alpha_{x,y,z} = \alpha_{w \otimes x,y,z} \cdot \alpha_{w,x,y \otimes z} \quad .$$

▶ **Definition 23** (`disp_bicat_univstrongfunctor`)**.** *The displayed bicategory* $\mathbf{Cat}_S$ *is the (non-full) displayed sub-bicategory of* $\mathbf{Cat}_P$ *where the displayed morphisms are proofs of the proposition*

$$\mathsf{isIso}(\epsilon) \times \prod_{x,y:\mathcal{C}} \mathsf{isIso}(\mu_{x,y}) \quad .$$

The bicategory of monoidal categories, lax (resp. strong) monoidal functors, and monoidal natural transformations is denoted by $\mathbf{MonCat} := \int \mathbf{Cat}_P$ (resp. $\mathbf{MonCat}^{stg} := \int \mathbf{Cat}_S$).

▶ Remark 24. The constructions are summarized in Figure 2. The dashed arrows correspond to the projection induced by the product of the displayed bicategories to any of the components. In particular, this means that the dashed arrows induce a (bi)pullback (of displayed bicategories). The filled arrows represent that we have a forgetful pseudofunctor (given by the projection of a total bicategory to its base bicategory). Lastly, the hooked arrows mean that the domain is constructed as a (displayed) full sub-bicategory.

▶ Remark 25. An object in $\mathbf{MonCat}$ is of the form $(((\mathcal{C}, \otimes, I), \lambda, \rho, \alpha), tri, pent)$. Usually, one wants to consider an object in $\mathbf{MonCat}$ to be of the form $(\mathcal{C}, (((\otimes, I), \lambda, \rho, \alpha), tri, pent))$, i. e., as a category equipped with a monoidal structure. The displayed bicategory whose objects are categories equipped with a monoidal structure can be constructed by applying the sigma construction ([1, Definition 6.6(2)], `sigma_bicat`). Furthermore, this displayed bicategory is univalent by a criterion presented in [1]. As this does not change the message of the paper, we refer the reader to [1] for the precise statements, but we do show that the criteria are satisfied in Lemmas 31, 33, and 35.

▶ Remark 26. In the formalization of $\mathbf{Cat}_{LU}$ (resp. $\mathbf{Cat}_{RU}$, $\mathbf{Cat}_{A}$), we do not yet require a left unitor (resp. right unitor, associator) to be an isomorphism. Since being an isomorphism is a proposition, we could and did add these three (indexed) conditions only in the formalization of $\mathbf{Cat}_{P}$. This simplifies the proof of univalence of the bicategory of univalent monoidal categories that is built from **MonCat**.

In Section 4, we construct a Rezk completion for monoidal categories. We are interested in studying the hom-categories of **MonCat** and thus, in particular, the displayed hom-categories. We now introduce some notations. Let $\mathcal{B}$ be a bicategory and $x, y : \mathcal{B}$ objects. The hom-category from x to y is denoted by $\mathcal{B}(x, y)$. Any morphism $f : \mathcal{B}(x, y)$ induces a functor between hom-categories, more precisely:

▶ **Definition 27.** *Let $\mathcal{B}$ be a bicategory, $f : \mathcal{B}(x, y)$ a morphism and $z : \mathcal{B}$ an object. The **functor given by precomposition with f and target object z** is the functor*

$$f \cdot (-) : \mathcal{B}(y, z) \to \mathcal{B}(x, z) \ ,$$

where the action on the objects is given by precomposition, i. e., $g \mapsto f \cdot g$, and the action on the morphisms is given by left whiskering, i. e., $\alpha \mapsto f \triangleleft \alpha$.

We also refer to the functor given by precomposition with f as the **precomposition functor with f**.

Let $\mathcal{D}$ be a displayed bicategory over $\mathcal{B}$ and $\bar{x} \in \mathcal{D}_x$ and $\bar{y} \in \mathcal{D}_y$ be displayed objects. The (total) hom-category $\int \mathcal{D}((x, \bar{x}), (y, \bar{y}))$ can be constructed as a total category of a displayed category over $\mathcal{B}(x, y)$. We denote this displayed category by $\mathcal{D}(\bar{x}, \bar{y})$ (so we use the same notation for the hom-categories and displayed hom-categories).

In particular, the precomposition functor w. r. t. the total bicategory $\int \mathcal{D}$ of a morphism $(f, \bar{f})$ can be defined as a displayed functor over the precomposition functor $f \cdot (-)$ (w. r. t. $\mathcal{B}$) where we precompose/left whisker (in the displayed sense) with $\bar{f}$:

▶ **Definition 28.** *Let $\mathcal{D}$ be a displayed bicategory over a bicategory $\mathcal{B}$, $\bar{x} : \mathcal{D}_x, \bar{y} : \mathcal{D}_y$ displayed objects, $\bar{f} : \mathcal{D}_f(\bar{x}, \bar{y})$ a displayed morphism and $\bar{z} : \mathcal{D}_z$ a displayed object. The **displayed functor given by precomposition with $\bar{f}$ and target displayed object $\bar{z}$** is the displayed functor*

$$\bar{f} \cdot (-) : \mathcal{D}(\bar{y}, \bar{z}) \to \mathcal{D}(\bar{x}, \bar{z})$$

over the functor given by precomposition with f and target object z.

We also refer to the displayed functor given by precomposition with $\bar{f}$ as the **displayed precomposition functor with $\bar{f}$**.

3 The univalent bicategory of monoidal categories

In this section we present our proof of univalence of the bicategory $\mathbf{MonCat}_{univ}$ of univalent monoidal categories, with Theorem 37 as the main result. (We also obtain a version with strong monoidal functors in place of lax monoidal functors.) In this proof, we rely heavily on the *displayed* machinery built in [1], for modular construction of bicategories, and proofs of their univalence.

In the formalization of this univalence proof, we have not used the formalization of a monoidal category as presented above. Instead, we have changed the definition of a tensor from being a functor to a more explicit, unfolded definition. It is not necessarily obvious that the resulting bicategory is indeed that of monoidal categories, lax (resp. strong) monoidal functors, and monoidal natural transformations. Therefore, we construct an equivalence of types of monoidal categories as presented above on the one hand and using this explicit definition on the other hand (`cmonoidal_to_noncurriedmonoidal`, `cmonoidal_adjequiv_noncurried_hom`).

Recall from Lemma 12 that the total bicategory of a displayed bicategory is univalent if both the base bicategory and the displayed bicategory are univalent. Since $\mathbf{Cat}_{univ}$ is univalent [[1, Prop. 3.19], `univalent_cat_is_univalent_2`], the task of proving $\mathbf{MonCat}_{univ}$ univalent therefore reduces to showing that $\Sigma_{\Sigma_{\mathbf{Cat}_{TU}}\mathbf{Cat}_{UA}}\mathbf{Cat}_P$ from the previous section is univalent, restricted to the full sub-bicategory $\mathbf{Cat}_{univ}$ of $\mathbf{Cat}$. (This is to be read modulo the repackaging hinted to in Remark 25.)

The sigma construction of univalent displayed bicategories is univalent provided that both displayed bicategories are locally groupoidal and locally propositional [[1, Prop. 7.9], `sigma_disp_univalent_2_with_props`]. The previously defined displayed bicategories are locally propositional since they either express an (indexed) equality of morphisms or the type of 2-cells is the unit type. Thus in this section, we show that the displayed bicategories from Section 2 are univalent and locally groupoidal.

▶ Remark 29. In this section we restrict the displayed bicategories to the bicategory $\mathbf{Cat}_{univ}$ of univalent categories. For example, the restriction of $\mathbf{Cat}_{TU}$ is considered as the pullback of the displayed bicategory $\mathbf{Cat}_{TU}$ along the inclusion of $\mathbf{Cat}_{univ}$ into $\mathbf{Cat}$. We denote the restriction of the displayed bicategory $\mathbf{Cat}_\ell$ by $\mathbf{Cat}_\ell|_{univ}$ for $\ell \in \{T, U, TU, LU, RU, A, UA, P, S\}$.

▶ **Lemma 30** (`tensor_disp_is_univalent_2`)**.** *$\mathbf{Cat}_T|_{univ}$ is univalent.*

Proof. $\mathbf{Cat}_T|_{univ}$ is locally univalent by a straightforward calculation, we therefore only discuss that it is globally univalent.

Let $\otimes_1, \otimes_2$ be two tensors on $\mathcal{C}$. We have to show that $\mathsf{idtoiso}^{2,0}_{\otimes_1,\otimes_2}$ is an equivalence of types. In order to show this, we factorize this function as follows:

$$\begin{tikzcd}
\otimes_1 = \otimes_2 \arrow[r, "\mathsf{idtoiso}^{2,0}_{\otimes_1,\otimes_2}"] \arrow[d, "\mathsf{idtoeq}"'] & \mathsf{DispAdjEquiv}(\otimes_1,\otimes_2) \\
\mathsf{tensorEq}(\otimes_1,\otimes_2) \arrow[r, "\mathsf{eqtoiso}"'] & \mathsf{tensorIso}(\otimes_1,\otimes_2) \arrow[u]
\end{tikzcd} ,$$

where $\mathsf{tensorEq}(\otimes_1,\otimes_2)$ is the type

$$\sum_{\alpha:\prod_{x,y:\mathcal{C}}, x\otimes_1 y = x \otimes_2 y} \ \prod_{f:\mathcal{C}(x_1,x_2)} \ \prod_{g:\mathcal{C}(y_1,y_2)} f \otimes_1 g = f \otimes_2 g \ ,$$

where the equality $f \otimes_1 g = f \otimes_2 g$ is dependent over α_{x_1,y_1} and α_{x_2,y_2}.

The type $\mathsf{tensorIso}(\otimes_1,\otimes_2)$ is the same as $\mathsf{tensorEq}(\otimes_1,\otimes_2)$ where we replaced the first equality by an isomorphism (and the dependent equality of morphisms is replaced by pre- and post-composing with the isomorphism).

The function $\mathsf{idtoeq} : \otimes_1 = \otimes_2 \to \mathsf{tensorEq}(\otimes_1,\otimes_2)$ maps equality to pointwise equality (on both the objects and morphisms). Because our hom-types are sets, this is an equivalence. The function $\mathsf{eqtoiso} : \mathsf{tensorEq}(\otimes_1,\otimes_2) \to \mathsf{tensorIso}(\otimes_1,\otimes_2)$ replaces identity by isomorphism. Since $\mathcal{C}$ is a univalent category, $\mathsf{eqtoiso}$ is indeed an equivalence. Since a displayed adjoint equivalence in $\mathbf{Cat}_T$ translates into the notion of $\mathsf{tensorIso}(\otimes_1,\otimes_2)$, we construct in a straightforward manner a function from $\mathsf{tensorIso}(\otimes_1,\otimes_2)$ to $\mathsf{DispAdjEquiv}(\otimes_1,\otimes_2)$, which is for the same reason an equivalence. ◀

Each type of (displayed) 2-cells in $\mathbf{Cat}_U$ is contractible, hence:

▶ **Lemma 31** (`tensor_disp_locally_groupoidal`). *$\mathbf{Cat}_T|_{univ}$ is locally groupoidal.*

Proof. $\mathbf{Cat}_T|_{univ}$ being locally groupoidal means that if a natural isomorphism α preserves the tensor, then so does its inverse. This is immediate since the tensor product of isomorphisms is again an isomorphism (by functoriality of the tensor). ◀

▶ **Lemma 32** (`unit_disp_is_univalent_2`). *$\mathbf{Cat}_U|_{univ}$ is univalent.*

Proof. $\mathbf{Cat}_U|_{univ}$ is locally univalent by a straightforward calculation. Therefore, we only discuss why it is globally univalent.

Let $I, J : \mathcal{C}$ be objects representing a unit object. As with the tensor layer, we factorize $\mathsf{idtoiso}^{2,0}_{\mathsf{I,J}}$ and show that each function in the factorization is an equivalence. The factorization is given by:

$$\begin{tikzcd}
I = J \arrow[rr, "\mathsf{idtoiso}^{2,0}_{\mathsf{I,J}}"] \arrow[dr] & & \mathsf{DispAdjEquiv}(I, J) \\
& I \cong J \arrow[ur] &
\end{tikzcd}$$

The definition of a displayed adjoint equivalence in this displayed bicategory translates precisely to an isomorphism in the underlying category $\mathcal{C}$, which gives us the arrow to the right and a proof that it is an equivalence. The left arrow is given by $\mathsf{idtoiso}_{I,J}$ and is an equivalence precisely because $\mathcal{C}$ is a univalent category. ◀

▶ **Lemma 33** (`unit_disp_locally_groupoidal`). *$\mathbf{Cat}_U|_{univ}$ is locally groupoidal.*

▶ **Lemma 34** (`assunitors_disp_is_univalent_2`). *$\mathbf{Cat}_{UA}|_{univ}$ is univalent.*

Proof. Since the product of univalent displayed bicategories is univalent, it remains to show that $\mathbf{Cat}_{LU}|_{univ}$,$\mathbf{Cat}_{RU}|_{univ}$ and $\mathbf{Cat}_A|_{univ}$ are univalent.

These displayed bicategories are locally univalent because the type of (displayed) 2-cells is the unit type and the type of (displayed) 1-cells is a proposition.

Since the type of objects (resp. morphisms, 2-cells) is a set (resp. proposition, contractible) and the base category is locally univalent, we can apply [1, Prop. 7.10]. This proposition asserts that a displayed bicategory is univalent if a function of type $(a \simeq_{\mathsf{idtoiso}^{2,0}(p)} b) \to (a =_p b)$ can be constructed. The latter means precisely that we have to construct displayed morphisms over an identity morphism. In the case of the left unitor, this means that we have to construct a term of type $(\lambda_1 = \lambda_2)$ provided that the identity morphism on $(\mathcal{C}, \otimes, I)$ preserves the left unitor (as in Definition 19.2). The proofs that $\mathbf{Cat}_{RU}|_{univ}$ and $\mathbf{Cat}_A|_{univ}$ are univalent is analogous. ◀

▶ **Lemma 35** (`assunitors_disp_locally_groupoidal`). *$\mathbf{Cat}_{UA}|_{univ}$ is locally groupoidal.*

Proof. This follows from the following lemmas:

1. The product of locally groupoidal displayed bicategories is locally groupoidal.
2. A displayed bicategory whose type of displayed 2-cells is the unit is locally groupoidal.

◀

A full displayed sub-bicategory of a univalent displayed bicategory is univalent, hence:

▶ **Lemma 36** (`tripent_disp_is_univalent_2`). *$\mathbf{Cat}_P|_{univ}$ is univalent.*

Since a full displayed sub-bicategory of a displayed locally groupoidal bicategory is locally groupoidal, we have that $\mathbf{Cat}_P|_{univ}$ is locally groupoidal.

▶ **Theorem 37** (`UMONCAT_is_univalent_2`). *The bicategory of univalent monoidal categories, lax monoidal functors, and monoidal natural transformations is univalent.*

▶ **Lemma 38** (`UMONCAT_disp_strong_is_univalent_2`). $\mathbf{Cat}_S|_{univ}$ *is univalent.*

Proof. This follows immediately from Lemma 36 since the type of displayed 1-cells is a mere proposition. ◀

▶ **Theorem 39** (`UMONCAT_strong_is_univalent_2`). *The bicategory of univalent monoidal categories, strong monoidal functors, and monoidal natural transformations is univalent.*

4 The Rezk completion for monoidal categories

Some constructions of (monoidal) categories do not yield univalent (monoidal) categories. For instance, categories built from syntax usually have *sets* of objects; the presence of non-trivial isomorphisms in such a category hence entails that it is not univalent. Another example is when constructing colimits of univalent monoidal categories; the usual construction of such a colimit often yields a non-univalent monoidal category. In such cases, a "completion operation", turning a monoidal category into a univalent one, is handy.

In this section we construct, for each monoidal category, a free univalent monoidal category, which we call the *monoidal Rezk completion*. More precisely, we solve the following problem:

▶ **Problem 40.** *Given a Rezk completion $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ of a category $\mathcal{C}$ and a monoidal structure $M := (\otimes, I, \lambda, \rho, \alpha)$ on $\mathcal{C}$, construct a monoidal structure $\hat{M} := (\hat{\otimes}, \hat{I}, \hat{\lambda}, \hat{\rho}, \hat{\alpha})$ on $\mathcal{D}$ and a strong monoidal structure for $\mathcal{H}$ w.r.t. M and $\hat{M}$, such that for any univalent monoidal category $(\mathcal{E}, N)$, the isomorphism of categories*

$$\mathcal{H} \cdot (-) : \mathbf{Cat}(\mathcal{D}, \mathcal{E}) \to \mathbf{Cat}(\mathcal{C}, \mathcal{E})$$

lifts to the category of lax (resp. strong) monoidal functors:

$$\mathcal{H} \cdot (-) : \mathbf{MonCat}((\mathcal{D}, \hat{M}), (\mathcal{E}, N)) \to \mathbf{MonCat}((\mathcal{C}, M), (\mathcal{E}, N)) \ .$$

Once solved, we call $(\mathcal{D}, \hat{M})$ the *monoidal Rezk completion of* $(\mathcal{C}, M)$. Analogous to the Rezk completion for categories, the monoidal Rezk completion exhibits the bicategory $\mathbf{MonCat}_{univ}$ (resp. $\mathbf{MonCat}^{stg}_{univ}$) as a reflective full sub-bicategory of $\mathbf{MonCat}$ (resp. $\mathbf{MonCat}^{stg}$).

Although any categorical structure on a category can be transported along an equivalence of categories such that they become equivalent in the corresponding bicategory of structured categories, this might not be the case if one considers a weak equivalence. On the way towards solving Problem 40, we show, in particular, how to transport a monoidal structure along a weak equivalence of categories (see Remark 61), provided that the target category is univalent. That construction is not limited to the specific weak equivalence given by the Rezk completion.

Analogous to the univalence proof of $\mathbf{MonCat}_{univ}$ (resp. $\mathbf{MonCat}^{stg}_{univ}$) given in Section 3, we rely on the theory of displayed categories in order to solve this problem by dividing it into subgoals. In each of the subgoals, we use the same strategy. In Section 4.1, we explain the strategy in detail for the subgoal of equipping $\mathcal{D}$ (resp. $\mathcal{H} : \mathcal{C} \to \mathcal{D}$) with a tensor (resp. tensor-preserving structure).

4.1 The Rezk completion of a category with a tensor

Let $\mathcal{C}$ be a category and $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ a Rezk completion of $\mathcal{C}$. Let $\otimes : \mathcal{C} \times \mathcal{C} \to \mathcal{C}$ be a functor.

In this section we equip $\mathcal{D}$ with a functor $\hat{\otimes} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ such that

1. $\mathcal{H}$ has the structure of a *strong tensor-preserving* functor, i. e., we have a natural isomorphism $\mu^{\mathcal{H}} : (\mathcal{H} \times \mathcal{H}) \cdot \hat{\otimes} \Rightarrow \otimes \cdot \mathcal{H}$.
2. The precomposition functor of $(\mathcal{H}, \mu^{\mathcal{H}})$ is an isomorphism of categories.

▶ **Definition 41** (`TransportedTensor`, `TransportedTensorComm`)**.** *The* lifted tensor $\hat{\otimes}$ *on* $\mathcal{D}$ *is the (unique) functor* $\hat{\otimes} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ *such that there is a natural isomorphism as depicted in the following diagram:*

$$\begin{tikzcd}
& \mathcal{D} \times \mathcal{D} \arrow[dr, "\hat{\otimes}"] \arrow[dd, Rightarrow, shorten=15pt, "\mu^{\mathcal{H}}"] & \\
\mathcal{C} \times \mathcal{C} \arrow[ur, "\mathcal{H} \times \mathcal{H}"] \arrow[dr, "\otimes"'] & & \mathcal{D} \\
& \mathcal{C} \arrow[ur, "\mathcal{H}"'] &
\end{tikzcd}$$

▶ Remark 42. The functor $\hat{\otimes}$ is given by applying Lemma 4 to the weak equivalence $\mathcal{H} \times \mathcal{H} : \mathcal{C} \times \mathcal{C} \to \mathcal{D} \times \mathcal{D}$.

▶ Remark 43. The natural isomorphism is labelled as $\mu^{\mathcal{H}}$ because this natural isomorphism is precisely the structure we need to have that $\mathcal{H}$ is a (strong) tensor-preserving functor.

▶ **Lemma 44** (`HT_eso`)**.** *Let* $\mathcal{E}$ *be a univalent category and* $\otimes_{\mathcal{E}} : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ *be a functor. The displayed precomposition functor (Definition 28)* $\mu^{\mathcal{H}} \cdot (-)$ *with target displayed object* $\otimes_{\mathcal{E}}$ *(as a displayed object in* $\mathbf{Cat}_T$*) is displayed split essentially surjective. Consequently, the precomposition functor*

$$(\mathcal{H}, \mu^{\mathcal{H}}) \cdot (-) : \textstyle\int \mathbf{Cat}_T\left((\mathcal{D}, \hat{\otimes}), (\mathcal{E}, \otimes_E)\right) \to \textstyle\int \mathbf{Cat}_T\left((\mathcal{C}, \otimes), (\mathcal{E}, \otimes_E)\right)$$

is essentially surjective.

Proof. Let $G : \mathcal{D} \to \mathcal{E}$ be a functor and $\mu^{\mathcal{H} \cdot G}$ a natural transformation of type

$$(\mathcal{H} \times \mathcal{H}) \cdot (G \times G) \cdot \otimes_{\mathcal{E}} \Rightarrow \otimes \cdot \mathcal{H} \cdot G \ .$$

witnessing that $\mathcal{H} \cdot G$ is a lax tensor-preserving functor. We have to construct a natural transformation witnessing that G is a lax tensor-preserving functor, i. e., we have to define a natural transformation

$$\mu^G : (G \times G) \cdot \otimes_{\mathcal{E}} \Rightarrow \hat{\otimes} \cdot G \ .$$

Since $\mathcal{H} \times \mathcal{H}$ is a weak equivalence and $\mathcal{E}$ is univalent, it suffices to define a natural transformation of type

$$(\mathcal{H} \times \mathcal{H}) \cdot (G \times G) \cdot \otimes_{\mathcal{E}} \Rightarrow (\mathcal{H} \times \mathcal{H}) \cdot \hat{\otimes} \cdot G \ .$$

which we define as:

$$\begin{tikzcd}
& \mathcal{D} \times \mathcal{D} \arrow[r, "G \times G"] & \mathcal{E} \times \mathcal{E} \arrow[dr, "\otimes_{\mathcal{E}}"] \arrow[dl, Rightarrow, shorten=12pt, "\mu^{\mathcal{H} \cdot G}"'] & \\
\mathcal{C} \times \mathcal{C} \arrow[ur, "\mathcal{H} \times \mathcal{H}"] \arrow[r, "\otimes"'] \arrow[dr, "\mathcal{H} \times \mathcal{H}"'] & \mathcal{C} \arrow[r, "\mathcal{H}"'] \arrow[d, Rightarrow, shorten=8pt, "(\mu^{\mathcal{H}})^{-1}"] & \mathcal{D} \arrow[r, "G"'] & \mathcal{E} \\
& \mathcal{D} \times \mathcal{D} \arrow[ur, "\hat{\otimes}"'] & &
\end{tikzcd}$$

For a detailed proof that $\mu^{\mathcal{H} \cdot G}$ is (displayed) isomorphic to the (displayed) composition of $\mu^{\mathcal{H}}$ and μ^G, we refer the reader to the formalization. ◀

▶ **Lemma 45** (HT_ff). *Let $\mathcal{E}$ be a univalent category and $\otimes_{\mathcal{E}} : \mathcal{E} \times \mathcal{E} \to \mathcal{E}$ be a functor. The displayed precomposition functor $\mu^{\mathcal{H}} \cdot (-)$ is displayed fully faithful. Consequently, the precomposition functor $(\mathcal{H}, \mu^{\mathcal{H}}) \cdot (-)$ between the tensor-preserving functor categories is fully faithful.*

Proof. It is displayed faithful because the type stating that a natural transformation preserves a tensor is a mere proposition. In order to show that it is displayed full, notice that we have to show an equality of morphisms, i. e., a proposition. Therefore, we are able to use that $\mathcal{H} \times \mathcal{H}$ is essentially surjective which allows us to work with objects in $\mathcal{C}$ instead of $\mathcal{D}$ which leads to the result. ◀

▶ **Theorem 46** (precomp_tensor_catiso). *A category equipped with a tensor admits a Rezk completion: Let $(\mathcal{E}, \otimes_{\mathcal{E}}) : \int \mathbf{Cat}_T$. If $\mathcal{E}$ is univalent, then*

$$(\mathcal{H}, \mu^{\mathcal{H}}) \cdot (-) : \int \mathbf{Cat}_T \left((\mathcal{D}, \hat{\otimes}), (\mathcal{E}, \otimes_E)\right) \to \int \mathbf{Cat}_T \left((\mathcal{C}, \otimes), (\mathcal{E}, \otimes_E)\right)$$

is an isomorphism of categories.

Proof. First notice that both categories are univalent, indeed: since $\mathcal{E}$ is univalent, so are $\mathbf{Cat}(\mathcal{D}, \mathcal{E})$ and $\mathbf{Cat}(\mathcal{C}, \mathcal{E})$ and in Section 3, we have proven that the displayed bicategory $\mathbf{Cat}_T|_{univ}$ is locally univalent, i. e., the displayed hom-categories are univalent. Hence, it suffices to show that this functor is a weak equivalence, i. e., fully faithful and essentially surjective. Fully faithfulness can always be concluded if both the functor on the base categories and the displayed functor are. The total functor is essentially surjective if this holds on the base and at the displayed level, provided extra information: it suffices that the base category and the displayed category are univalent. So we conclude the result from combining the assumption that $\mathcal{H}$ is a weak equivalence and lemmas 45 and 44. ◀

▶ Remark 47. The strategy introduced in this section will be repeated in the next section, so we refer back to this section for the necessary details (if needed).

4.2 The Rezk completion of a category with a tensor and unit

In Section 4.1, we have shown how the structure of a tensor $\otimes$ on $\mathcal{C}$ transports along a weak equivalence $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ to a tensor on a univalent category $\mathcal{D}$. Furthermore, $\mathcal{H}$ has the structure of a strong tensor-preserving functor and that $(\mathcal{D}, \hat{\otimes})$ is universal in the sense that objects in $\int \mathbf{Cat}_T$ admit a Rezk completion.

In this section, we show that the same result holds when we add the choice of an object to a category, playing the role of the tensorial unit. This construction is trivial, but we will also discuss how we can conclude that objects in $\int \mathbf{Cat}_{TU}$ admit a Rezk completion.

As before, let $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ be a weak equivalence from a category $\mathcal{C}$ to a univalent category $\mathcal{D}$. Let $I : \mathcal{C}$, thus $(\mathcal{C}, I) : \int \mathbf{Cat}_U$. Clearly we have $(\mathcal{H}, \mathsf{Id}_{\mathcal{H}\, I}) : \int \mathbf{Cat}_U((\mathcal{C}, I), (\mathcal{D}, \mathcal{H}\, I))$.

To conclude that $(\mathcal{D}, \mathcal{H}\, I)$ is universal, we apply the same reasoning as in Section 4.1. We have to show that for any $(\mathcal{E}, I_{\mathcal{E}}) : \mathbf{Cat}_U$ with $\mathcal{E}$ univalent, the displayed precomposition functor

$$\mathsf{Id}_{\mathcal{H}\, I} \cdot (-) : \mathbf{Cat}_U(\mathcal{H}\, I, I_{\mathcal{E}}) \to \mathbf{Cat}_U(I, I_{\mathcal{E}})$$

is displayed fully faithful and displayed split essentially surjective. We denote $\hat{I} := (\mathcal{H}\, I)$ and $\epsilon^{\mathcal{H}} := \mathsf{Id}_{\hat{I}}$.

▶ **Lemma 48** (`HU_eso`). *The displayed precomposition functor (Definition 28) $\epsilon^{\mathcal{H}} \cdot (-)$ with target displayed object $I_{\mathcal{E}}$ is displayed split essentially surjective. Consequently, the precomposition functor $(\mathcal{H}, \epsilon^{\mathcal{H}}) \cdot (-)$ with target object $(\mathcal{E}, I_{\mathcal{E}})$ between unit tensor-preserving functor categories is essentially surjective.*

Proof. It is merely surjective since the witness, expressing that the weak equivalence preserves the unit, is an identity morphism. ◀

▶ **Lemma 49** (`HU_ff`). *The displayed precomposition functor $\epsilon^{\mathcal{H}} \cdot (-)$ is displayed fully faithful. Consequently, the precomposition functor $(\mathcal{H}, \epsilon^{\mathcal{H}}) \cdot (-)$ between the unit-preserving functor categories is fully faithful.*

Proof. It is displayed faithful since the type of 2-cells is a property. The witness expressing that the weak equivalence preserves the unit is an identity morphism. Hence, it is displayed full. ◀

Using the exact same reasoning used in Theorem 46, we conclude:

▶ **Theorem 50** (`precomp_unit_catiso`). *A category equipped with a unit admits a Rezk completion: Let $(\mathcal{E}, I_{\mathcal{E}}) : \int \mathbf{Cat}_U$. If $\mathcal{E}$ is univalent, then*

$$(\mathcal{H}, \epsilon^{\mathcal{H}}) \cdot (-) : \textstyle\int \mathbf{Cat}_U \left((\mathcal{D}, \hat{I}), (\mathcal{E}, I_{\mathcal{E}})\right) \to \int \mathbf{Cat}_U \left((\mathcal{C}, I), (\mathcal{E}, I_{\mathcal{E}})\right)$$

is an isomorphism of categories.

So we have proven that objects in $\mathbf{Cat}_T$ and $\mathbf{Cat}_U$ admit a Rezk completion. From these results, we conclude that objects in $\mathbf{Cat}_{TU}$ admit a Rezk completion:

▶ **Theorem 51** (`precomp_tensorunit_catiso`). *Let $(\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}}) : \mathbf{Cat}_{TU}$. If $\mathcal{E}$ is univalent, then*

$$(\mathcal{H}, \mu^{\mathcal{H}}, \epsilon^{\mathcal{H}}) \cdot (-) : \textstyle\int \mathbf{Cat}_{TU} \left((\mathcal{D}, \hat{\otimes}, \hat{I}), (\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}})\right) \to \int \mathbf{Cat}_{TU} \left((\mathcal{C}, \otimes, I), (\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}})\right)$$

is an isomorphism of categories, i. e., objects in $\int \mathbf{Cat}_{TU}$ admit a Rezk completion.

Proof. The product of univalent displayed bicategories is again univalent. Thus, both the domain and codomain of this functor are univalent. Hence, by the same argument as in Theorem 46, it reduces to proving that the displayed precomposition functor is a displayed weak equivalence. The displayed precomposition functor is the product of the displayed precomposition functors of $\mu^{\mathcal{H}}$ resp. $\epsilon^{\mathcal{H}}$. Since the product of displayed weak equivalences is again a weak equivalence, the result now follows. ◀

4.3 The Rezk completion of a category with a tensor, unit, unitors and associator

In this section, we prove that every object in $\int \mathbf{Cat}_{LU}$ (resp. $\int \mathbf{Cat}_{RU}$ and $\int \mathbf{Cat}_{A}$) has a Rezk completion.

As above, we let $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ be a weak equivalence from a category $\mathcal{C}$ to a univalent category $\mathcal{D}$, and let $\mathcal{C}$ be equipped with a tensor $\otimes$ and a unit I. The lifted tensor on $\mathcal{D}$ is denoted by $\hat{\otimes}$ and $\hat{I} := \mathcal{H}\, I$. The witness that $\mathcal{H}$ preserves the tensor (resp. unit) (strongly) is denoted by $\mu^{\mathcal{H}}$ (resp. $\epsilon^{\mathcal{H}} = \mathsf{Id}_{\mathcal{H}\, I}$).

▶ Remark 52. In all the constructions of this section, we use the lifted tensor $\hat{\otimes}$ and unit $\hat{I}$. The specific shape of these lifts does not matter; we could state the constructions for an *arbitrary* Rezk completion of $\mathbf{Cat}_{TU}$. However, by univalence we have uniqueness of the tensor and unit on $\mathcal{D}$ under the proviso that $\mathcal{H}$ preserves them both.

Before lifting a left unitor from $\mathcal{C}$ to $\mathcal{D}$, we first define a natural isomorphism witnessing that the weak equivalence preserves tensoring with the unit object (on the left):

▶ **Lemma 53** (LiftPreservesPretensor). *There is a natural isomorphism $\mathcal{H} \cdot (\hat{I} \hat{\otimes} -) \Rightarrow (I \otimes -) \cdot \mathcal{H}$.*

Proof. This is given by the following composition:

$$\begin{tikzcd}
\mathcal{C} \arrow[r, "\mathcal{H}"] \arrow[d, "{(I,-)}"'] & \mathcal{D} \arrow[d, "{(\hat{I},-)}"] \\
\mathcal{C} \times \mathcal{C} \arrow[r, "\mathcal{H} \times \mathcal{H}"] \arrow[d, "\otimes"'] & \mathcal{D} \times \mathcal{D} \arrow[d, "\hat{\otimes}"] \\
\mathcal{C} \arrow[r, "\mathcal{H}"'] & \mathcal{D}
\arrow[from=2-1, to=3-2, phantom, "\mu^{\mathcal{H}} \Downarrow"]
\end{tikzcd}$$

where the upper square is given by a trivial equality of functors. ◀

▶ **Definition 54** (TransportedLeftUnitor). *Let λ be a left unitor on $(\mathcal{C}, \otimes, I)$, that is, $(\mathcal{C}, \otimes, I, \lambda) : \int \mathbf{Cat}_{LU}$. The* lifted left unitor *$\hat{\lambda}$ on $(\mathcal{D}, \hat{\otimes}, \hat{I})$ is the unique natural isomorphism that maps to the vertical composition of the natural isomorphism (defined in Lemma 53) and $\lambda \triangleright \mathcal{H}$, under the precomposition functor with $\mathcal{H}$.*

An immediate calculation shows:

▶ **Lemma 55** (H_plu). *$\mathcal{H}$ preserves the left unitor.*

▶ **Theorem 56** (precomp_lunitor_catiso). *The objects in $\int \mathbf{Cat}_{LU}$ admit a Rezk completion:*

Let $(\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}}, \lambda_{\mathcal{E}}) : \int \mathbf{Cat}_{LU}$. If $\mathcal{E}$ is univalent, then $(\mathcal{H}, \mu^{\mathcal{H}}, \epsilon^{\mathcal{H}}, \mathsf{plu}^{\mathcal{H}}) \cdot (-)$ of type

$$\textstyle\int \mathbf{Cat}_{LU}\left((\mathcal{D}, \hat{\otimes}, \hat{I}, \hat{\lambda}), (\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}}, \lambda_{\mathcal{E}})\right) \to \int \mathbf{Cat}_{LU}\left((\mathcal{C}, \otimes, I, \lambda), (\mathcal{E}, \otimes_{\mathcal{E}}, I_{\mathcal{E}}, \lambda_{\mathcal{E}})\right)$$

is an isomorphism of categories, where $\mathsf{plu}^{\mathcal{H}}$ is a witness that $\mathcal{H}$ preserves the left unitor (as provided by Lemma 55).

Proof. As before, it reduces to show that the displayed precomposition functor (Definition 28) is a displayed weak equivalence. It is displayed fully faithful since the type of 2-cells in $\mathbf{Cat}_{LU}$ is the unit type. We now show that it is displayed split essentially surjective. Let $G : \mathcal{D} \to \mathcal{E}$ be a lax tensor and unit preserving functor such that $\mathcal{H} \cdot G$ preserves the left unitor. We have to show that G also preserves the left unitor. Since we have to show a proposition, the claim now follows from combining the essential surjectivity of $\mathcal{H}$ and then applying the assumption on $\mathcal{H} \cdot G$. ◀

Completely analogous is the case of right unitor:

▶ **Theorem 57** (precomp_runitor_catiso). *The objects in $\int \mathbf{Cat}_{RU}$ admit a Rezk completion.*

In order to prove that every object in $\int \mathbf{Cat}_A$ has a Rezk completion, we use an analogous trick as is used for objects in, e. g., $\int \mathbf{Cat}_{LU}$. An associator for $(\mathcal{D}, \hat{\otimes})$ is a natural isomorphism between functors of type $(\mathcal{D} \times \mathcal{D}) \times \mathcal{D} \to \mathcal{D}$. Since the product of weak equivalences is again a weak equivalence, such a natural isomorphism corresponds uniquely to a natural isomorphism between functors of type $(\mathcal{C} \times \mathcal{C}) \times \mathcal{C} \to \mathcal{D}$. Analogous to the constructions of the left and right unitor, the natural isomorphism (of type $(\mathcal{C} \times \mathcal{C}) \times \mathcal{C} \to \mathcal{D}$) is not given by $\alpha \triangleright \mathcal{H}$ as this does not give us the correct type of functors. In the case of the left unitor, we only had to provide a natural isomorphism to match the domain, but for the associator, we furthermore need a natural isomorphism to match the codomain.

▶ **Theorem 58** (`precomp_associator_catiso`). *The objects in $\int \mathbf{Cat}_A$ admit a Rezk completion.*

4.4 The Rezk completion of a monoidal category

In this section, we are able to conclude that the objects in **MonCat** and $\mathbf{MonCat}^{stg}$ admit a Rezk completion.

In the previous sections, we have lifted all the structure of a monoidal category to a weakly equivalent univalent category.

However, it still remains to show that the lifted structure $(\mathcal{D}, \hat{\otimes}, \hat{I}, \hat{\lambda}, \hat{\rho}, \hat{\alpha})$ satisfies the properties of a monoidal category if $(\mathcal{C}, \otimes, I, \lambda, \rho, \alpha)$ does.

▶ **Lemma 59** (`TransportedTriangleEq`, `TransportedPentagonEq`). *The lifted monoidal structure satisfies the pentagon and triangle equalities: If the triangle (resp. pentagon) equality holds for $(\mathcal{C}, \otimes, I, \lambda, \rho, \alpha)$, then it also holds for $(\mathcal{D}, \hat{\otimes}, \hat{I}, \hat{\lambda}, \hat{\rho}, \hat{\alpha})$.*

▶ **Theorem 60** (`precomp_monoidal_catiso`). *Any monoidal category admits a Rezk completion (considered in the bicategory of lax monoidal functors).*

Proof. In Theorem 56, Theorem 57 and Theorem 58 we have shown how the categories $\int \mathbf{Cat}_{LU}$, $\int \mathbf{Cat}_{RU}$ and $\int \mathbf{Cat}_A$ admit a Rezk completion. Hence, $\int(\mathbf{Cat}_{LU} \times \mathbf{Cat}_{RU} \times \mathbf{Cat}_A)$ admits a Rezk completion.

Thus, to conclude that the total bicategory of $\mathbf{Cat}_P$ (over $\int(\mathbf{Cat}_{LU} \times \mathbf{Cat}_{RU} \times \mathbf{Cat}_A)$) admits a Rezk completion, it suffices to show that the displayed precomposition functor with respect to $\mathbf{Cat}_P$ is displayed fully faithful and displayed split essentially surjective. The displayed hom-categories of $\mathbf{Cat}_P$ are the terminal categories. Hence, the displayed precomposition functor must be the displayed identity functor. Consequently, this displayed precomposition functor is a weak equivalence. ◀

▶ Remark 61 (`RezkCompletion_monoidal_cat`, `RezkCompletion_monoidal_functor`). As part of the proof of Theorem 60, we have shown how to transfer a monoidal structure along a weak equivalence of categories, provided that the target category is univalent. More precisely, for any monoidal category $\mathcal{C}$, univalent category $\mathcal{D}$, and weak equivalence $\mathcal{H} : \mathcal{C} \to \mathcal{D}$, we construct a monoidal structure M on $\mathcal{D}$, and a structure of a (strong) monoidal functor on $\mathcal{H}$ with respect to $\mathcal{C}$ and M.

Next, we prove that any monoidal category admits a Rezk completion in the bicategory of strong monoidal functors. Concretely, we show the following theorem:

▶ **Theorem 62** (`precomp_strongmonoidal_catiso`). *Let $\mathcal{C}$ be a monoidal category and $\mathcal{H} : \mathcal{C} \to \mathcal{D}$ the Rezk completion of $\mathcal{C}$ as constructed in Theorem 60. If $\mathcal{E}$ is a univalent monoidal category, then*

$$\mathcal{H} \cdot (-) : \mathbf{MonCat}^{stg}(\mathcal{D}, \mathcal{E}) \to \mathbf{MonCat}^{stg}(\mathcal{C}, \mathcal{E})$$

is an isomorphism of categories.

Proof. First note that $\mathcal{H}$ is indeed strong monoidal by the definition of $\mu^{\mathcal{H}}$ and $\epsilon^{\mathcal{H}}$. Hence, the statement is well-defined.

As before, we have to conclude that the displayed precomposition functor (Definition 28) $((\mu^{\mathcal{H}})^{-1}, (\epsilon^{\mathcal{H}})^{-1}) \cdot (-)$ is fully faithful and displayed split essentially surjective.

The displayed precomposition functor is fully faithful since every type of displayed 2-cells in $\mathbf{MonCat}^{stg}$ is the unit type.

The displayed precomposition functor is split essentially surjective since the lift of a natural isomorphism is a natural isomorphism. ◀

4.5 The Rezk completion of a monoidal category using Day convolution

A concrete implementation of the Rezk completion of a category $\mathcal{C}$ is given by restricting the Yoneda embedding to its full image [2, Thm. 8.5]. It is well-known that any monoidal structure on $\mathcal{C}$ induces a monoidal structure on its category of presheaves $[\mathcal{C}^{op}, \mathbf{Set}]$ [8, Prop. 4.1]. The tensor product of two presheaves F, G is given by the *Day convolution* $F \otimes_{\mathsf{Day}} G$. Furthermore, the Day convolution of representable presheaves is again representable, i. e., for any two objects $x, y : \mathcal{C}$, one can construct a natural isomorphism

$$\mathcal{C}(-, x) \otimes_{\mathsf{Day}} \mathcal{C}(-, y) \cong \mathcal{C}(-, x \otimes y) \ .$$

Consequently, the Yoneda embedding has the structure of a strong monoidal functor. As one would expect, the full subcategory of representable presheaves becomes the monoidal Rezk completion. One way to show this result is to show that the universal property of monoidal Rezk completion holds. However, we already know that the full subcategory of representable presheaves has a monoidal structure (induced by the monoidal Rezk completion). Therefore, it suffices to show that the *Rezk monoidal structure* is equal to the *Day monoidal structure*.

Each piece of data of the *Rezk monoidal structure* is defined using a *universal property* in the sense that it is a unique lifting of some functor or natural transformation. For example, the (lifted) tensor product $\hat{\otimes}$ is the unique functor satisfying the equation

$$\otimes \cdot よ = (よ \times よ) \cdot \hat{\otimes} \ ,$$

where $よ$ is the Yoneda embedding restricted to its full image, i. e., the concrete weak equivalence. Using that a category of presheaves is univalent, the Day tensor product also satisfies this equation. Hence, the Day tensor product and the lifted tensor coincide. The lifted unit is by definition equal to the unit of the *Day monoidal structure*. Analogously, one can argue that the Day unitors and associator also satisfy the *universal property* of the lifted unitors resp. associator.

This shows that, for the concrete implementation of the Rezk completion using representable presheaves, the monoidal Rezk completion is given by the Day convolution.

▶ Remark 63. This section has briefly explained what one needs to do in order to work with a specific implementation of the Rezk completion of a category. Indeed, Let $(\mathcal{C}, \otimes, I, \lambda, \rho, \alpha)$ be a monoidal category and a *specific* univalent category $\mathcal{D}$ which is weakly equivalent to $\mathcal{C}$ as witnessed by $\mathcal{H} : \mathcal{C} \to \mathcal{D}$. Furthermore, assume we have a functor $\hat{\otimes} : \mathcal{D} \times \mathcal{D} \to \mathcal{D}$ and natural isomorphisms $\hat{\lambda}, \hat{\rho}$ and $\hat{\alpha}$ which have the types of a left unitor, right unitor and the associator (w. r. t. $\hat{\otimes}$ as the tensor and $\mathcal{H}\, I$ as the unit).

Then, in order to show that $(\mathcal{D}, \hat{\otimes}, \mathcal{H}\, I, \hat{\lambda}, \hat{\rho}, \hat{\alpha})$ is the monoidal Rezk completion, it suffices to show that the pieces of data satisfy the property of the lifted tensor, lifted left unitor, lifted right unitor and the lifted associator. In particular, one *does not* have to show manually that $(\mathcal{D}, \hat{\otimes}, \mathcal{H}\, I, \hat{\lambda}, \hat{\rho}, \hat{\alpha})$ is a monoidal category, $\mathcal{H}$ becomes a (strong) monoidal functor and that it satisfies the universal property of the monoidal Rezk completion; this all follows from the argument above.

5 Conclusion

We have studied (the bicategory of) monoidal categories in univalent foundations. First, we showed that the bicategory of univalent monoidal categories is univalent. Second, we constructed a Rezk completion for monoidal categories; specifically, we lifted the Rezk completion for categories to the monoidal structure. Our technique also works for lax and oplax monoidal categories, with minimal modifications. We have not presented this work here, but the UniMath code is available online.[2]

The second result provides a blueprint for constructing completion operations for "categories with structure". By "structure", we mean categorical structure such as functors and natural transformations. Here, the main challenge is to define a suitable notion of signature that allows us to specify structure on a category. Such a signature should translate into a suitable "tower" of displayed (bi)categories and come with the necessary boilerplate code for using it. Work on this topic will be reported elsewhere.

References

1 Benedikt Ahrens, Dan Frumin, Marco Maggesi, Niccolò Veltri, and Niels van der Weide. Bicategories in univalent foundations. *Math. Struct. Comput. Sci.*, 31(10):1232–1269, 2021. doi:10.1017/S0960129522000032.

2 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Struct. Comput. Sci.*, 25(5):1010–1039, 2015. doi:10.1017/S0960129514000486.

3 Benedikt Ahrens and Peter LeFanu Lumsdaine. Displayed categories. *Log. Methods Comput. Sci.*, 15(1), 2019. doi:10.23638/LMCS-15(1:20)2019.

4 Benedikt Ahrens, Paige Randall North, Michael Shulman, and Dimitris Tsementzis. The Univalence Principle, 2021. To be published in Memoirs of the American Mathematical Society. arXiv:2102.06275.

5 Johannes Schipp von Branitz and Ulrik Buchholtz. Using displayed univalent graphs to formalize higher groups in univalent foundations, 2021. URL: https://ulrikbuchholtz.dk/durgs.pdf.

6 Lucas Dixon and Aleks Kissinger. Monoidal categories, graphical reasoning, and quantum computation, 2009. Presented at Workshop on Computer Algebra Methods and Commutativity of Algebraic Diagrams (CAM-CAD). URL: https://www.researchgate.net/publication/265098183_Monoidal_Categories_Graphical_Reasoning_and_Quantum_Computation.

7 Pau Enrique Moliner, Chris Heunen, and Sean Tull. Space in monoidal categories. *Electronic Proceedings in Theoretical Computer Science*, 266, April 2017. doi:10.4204/EPTCS.266.25.

8 Geun Bin Im and G.M. Kelly. A universal property of the convolution monoidal structure. *Journal of Pure and Applied Algebra*, 43(1):75–88, 1986. doi:10.1016/0022-4049(86)90005-8.

9 Krzysztof Kapulkin and Peter LeFanu Lumsdaine. The simplicial model of Univalent Foundations (after Voevodsky). *Journal of the European Mathematical Society*, 23(6):2071–2126, 2021. URL: https://ems.press/journals/jems/articles/274693.

10 Yuri I. Manin and Matilde Marcolli. Homotopy theoretic and categorical models of neural information networks, 2020. arXiv:2006.15136.

11 Chad Nester. Concurrent process histories and resource transducers. *Log. Methods Comput. Sci.*, 19(1), 2023. doi:10.46298/lmcs-19(1:7)2023.

12 Dario Stein and Sam Staton. Compositional semantics for probabilistic programs with exact conditioning. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS*

[2] https://github.com/Kfwullaert/UniMath/tree/LaxMonoidalRezkCompletion

2021, Rome, Italy, June 29 - July 2, 2021, pages 1–13. IEEE, 2021. doi:10.1109/LICS52264.2021.9470552.

13 Kirk Sturtz. Categorical probability theory, 2014. arXiv:1406.6030.

14 The Coq Development Team. The Coq proof assistant, version 8.13.0, January 2021. URL: https://zenodo.org/record/4501022.

15 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book, Institute for Advanced Study, 2013.

16 Niels van der Weide. *Constructing Higher Inductive Types*. PhD thesis, Radboud University, 2020. URL: https://hdl.handle.net/2066/226923.

17 Vladimir Voevodsky, Benedikt Ahrens, Daniel Grayson, et al. UniMath – A computer-checked library of univalent mathematics. Available at http://unimath.github.io/UniMath/ , 2023.