

27th International Conference on Types for Proofs and Programs

TYPES 2021, June 14–18, 2021, Leiden, The Netherlands
(Virtual Conference)

Edited by

Henning Basold
Jesper Cockx
Silvia Ghilezan



Editors

Henning Basold 

LIACS, Leiden University, The Netherlands
h.basold@liacs.leidenuniv.nl

Jesper Cockx 

TU Delft, The Netherlands
J.G.H.Cockx@tudelft.nl

Silvia Ghilezan 

University of Novi Sad, Serbia
Mathematical Institute SASA, Belgrade, Serbia
gsilvia@uns.ac.rs

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 978-3-95977-254-9

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/978-3-95977-254-9>.

Publication date

August, 2022

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>.

License

This work is licensed under a Creative Commons Attribution 4.0 International license (CC-BY 4.0): <https://creativecommons.org/licenses/by/4.0/legalcode>.



In brief, this license authorizes each and everybody to share (to copy, distribute and transmit) the work under the following conditions, without impairing or restricting the authors' moral rights:

- Attribution: The work must be attributed to its authors.

The copyright is retained by the corresponding authors.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2021.0

ISBN 978-3-95977-254-9

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

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.

Editorial Board

- Luca Aceto (*Chair*, Reykjavik University, IS and Gran Sasso Science Institute, IT)
- Christel Baier (TU Dresden, DE)
- Mikolaj Bojanczyk (University of Warsaw, PL)
- Roberto Di Cosmo (Inria and Université de Paris, FR)
- Faith Ellen (University of Toronto, CA)
- Javier Esparza (TU München, DE)
- Daniel Král' (Masaryk University - Brno, CZ)
- Meena Mahajan (Institute of Mathematical Sciences, Chennai, IN)
- Anca Muscholl (University of Bordeaux, FR)
- Chih-Hao Luke Ong (University of Oxford, GB)
- Phillip Rogaway (University of California, Davis, US)
- Eva Rotenberg (Technical University of Denmark, Lyngby, DK)
- Raimund Seidel (Universität des Saarlandes, Saarbrücken, DE and Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Wadern, DE)

ISSN 1868-8969

<https://www.dagstuhl.de/lipics>

■ Contents

Preface	
<i>Henning Basold, Jesper Cockx, and Silvia Ghilezan</i>	0:vii
Papers	
Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control	
<i>Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer</i>	1:1–1:25
Formalisation of Dependent Type Theory: The Example of CaTT	
<i>Thibaut Benjamin</i>	2:1–2:21
Strictification of Weakly Stable Type-Theoretic Structures Using Generic Contexts	
<i>Rafaël Bocquet</i>	3:1–3:23
A Machine-Checked Proof of Birkhoff’s Variety Theorem in Martin-Löf Type Theory	
<i>William DeMeo and Jacques Carette</i>	4:1–4:21
Principal Types as Lambda Nets	
<i>Pietro Di Gianantonio and Marina Lenisa</i>	5:1–5:23
Internal Strict Propositions Using Point-Free Equations	
<i>István Donkó and Ambrus Kaposi</i>	6:1–6:21
Constructive Cut Elimination in Geometric Logic	
<i>Giulio Fellin, Sara Negri, and Eugenio Orlandelli</i>	7:1–7:16
A Succinct Formalization of the Completeness of First-Order Logic	
<i>Asta Halkjær From</i>	8:1–8:24
Simulating Large Eliminations in Cedille	
<i>Christa Jenkins, Andrew Marmaduke, and Aaron Stump</i>	9:1–9:22
Quantitative Polynomial Functors	
<i>Georgi Nakov and Fredrik Nordvall Forsberg</i>	10:1–10:22
Types and Terms Translated: Unrestricted Resources in Encoding Functions as Processes	
<i>Joseph W. N. Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez</i>	11:1–11:24
Size-Based Termination for Non-Positive Types in Simply Typed Lambda-Calculus	
<i>Yuta Takahashi</i>	12:1–12:23



■ Preface

This volume constitutes the post-proceedings of the *27th International Conference on Types for Proofs and Programs, TYPES 2021*, that was held virtually from 14 to 18 June 2021 with organisation based in Leiden. The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalised and computer assisted reasoning and computer programming. The meetings from 1990 to 2008 were annual workshops of a sequence of five EU-funded networking projects. Since 2009, TYPES has been run as an independent conference series. Previous TYPES meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Bergen 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) and Turin (2020). 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 of open and informal character. Selection of contributed talks is based on short abstracts; reporting work in progress and work presented or published elsewhere is welcome. A formal post-proceedings volume is prepared after the conference; papers submitted to that volume must represent unpublished work and are subjected to a full peer-review process.

Already in 2020, TYPES could not take place due to the situation surrounding SARS-CoV-2 and this very same situation forced us to make TYPES 2021 a virtual event. Given these circumstances and the general impact of measures taken in reaction to SARS-CoV-2, the conference and this volume can be considered a relative success. The conference programme consisted of four invited talks by Ulrik Buchholtz, Stephanie Balzer, Sara Negri and Pierre-Marie Pédrot. For a virtual event, the participation in the conference was high with 44 contributed talks, and 160 registered and about 70 active participants. The abstracts can be found online: <https://types21.liacs.nl/download/types-2021-book-of-abstracts/>

Even though the circumstances were not ideal, we were able to strengthen the link between the conference and these post-proceedings by giving more room for discussion during the conference of work that the authors intended to submit to the post-proceedings. Indeed, initially 23 papers were submitted to the post-proceedings, out of which 8 were retracted. However, we are happy to ultimately have post-proceedings consisting of 12 high-quality papers on formalised mathematics and semantics, foundations of type theory, geometric and linear logic, categorical methods in type theory, and types for processes. We thank all the authors and reviewers for their hard work to make this possible! Finally, we would like to thank the Leiden Institute of Advanced Computer Science for kindly covering the costs of the conference and the post-proceedings. This simplified participation given the virtual setup greatly! For the future, we hope that TYPES is not forced to be held again virtually, even though virtual attendance to talks is an interesting option to have, and the most important elements of TYPES can be brought back: discussion and spontaneous interaction!

Henning Basold, Jesper Cockx and Silvia Ghilezan, June 2022

27th International Conference on Types for Proofs and Programs (TYPES 2021).
Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan



Leibniz International Proceedings in Informatics
LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Verification of Bitcoin Script in Agda Using Weakest Preconditions for Access Control

Fahad F. Alhabardi ✉ 

Dept. of Computer Science, Swansea University, UK

Arnold Beckmann ✉ 

Dept. of Computer Science, Swansea University, UK

Bogdan Lazar ✉

University of Bath, UK

Anton Setzer ✉ 

Dept. of Computer Science, Swansea University, UK

Abstract

This paper contributes to the verification of programs written in Bitcoin’s smart contract language SCRIPT in the interactive theorem prover Agda. It focuses on the security property of access control for SCRIPT programs that govern the distribution of Bitcoins. It advocates that *weakest preconditions* in the context of Hoare triples are the appropriate notion for verifying access control. It aims at obtaining human-readable descriptions of weakest preconditions in order to close the validation gap between user requirements and formal specification of smart contracts.

As examples for the proposed approach, the paper focuses on two standard SCRIPT programs that govern the distribution of Bitcoins, *Pay to Public Key Hash (P2PKH)* and *Pay to Multisig (P2MS)*. The paper introduces an operational semantics of the SCRIPT commands used in P2PKH and P2MS, which is formalised in the Agda proof assistant and reasoned about using Hoare triples. Two methodologies for obtaining human-readable descriptions of weakest preconditions are discussed: (1) a step-by-step approach, which works backwards instruction by instruction through a script, sometimes grouping several instructions together; (2) symbolic execution of the code and translation into a nested case distinction, which allows to read off weakest preconditions as the disjunction of conjunctions of conditions along accepting paths. A syntax for equational reasoning with Hoare Triples is defined in order to formalise those approaches in Agda.

2012 ACM Subject Classification Theory of computation → Hoare logic; Theory of computation → Type theory; Theory of computation → Programming logic; Theory of computation → Interactive proof systems; Theory of computation → Operational semantics; Theory of computation → Denotational semantics; Security and privacy → Access control; Security and privacy → Logic and verification; Applied computing → Digital cash

Keywords and phrases Blockchain, Cryptocurrency, Bitcoin, Agda, Verification, Hoare logic, Bitcoin Script, P2PKH, P2MS, Access control, Weakest precondition, Predicate transformer semantics, Provable correctness, Symbolic execution, Smart contracts

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.1

Related Version *Full Version:* <https://arxiv.org/abs/2203.03054>

Supplementary Material *Software (Agde Source Code):*

<https://github.com/fahad19851ab/Smart--Contracts--Verification--With--Agda>
archived at [swh:1.dir:f552dd7468ce4fa08c193e2016b6c0e7580f1791](https://swh.1.dir:f552dd7468ce4fa08c193e2016b6c0e7580f1791)

Funding *Fahad F. Alhabardi:* Supported by Saudi Arabia Cultural Bureau in London.

Anton Setzer: Supported by COST actions CA20111 EuroProofNet and CA15123 EU Types.

Acknowledgements We would like to thank the anonymous referees for valuable comments and suggestions.



© Fahad F. Alhabardi, Arnold Beckmann, Bogdan Lazar, and Anton Setzer;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 1; pp. 1:1–1:25

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1 Introduction

Bitcoin, the first cryptocurrency, was introduced in 2008 by Satoshi Nakamoto [38] to provide a public payment mechanism, the blockchain, using pseudonymous keys in a peer-to-peer network with distributed control. Many other cryptocurrencies have been introduced since.

Bitcoin’s blockchain provides a scripting system for transactions called `SCRIPT`. Lists of instructions in `SCRIPT` are denoted *Bitcoin scripts* or simply *scripts*. The invention of Ethereum [13] strengthened Bitcoin by adding full (Turing complete) smart contracts to blockchains. In this context, smart contracts can be seen as programs that automatically execute when transactions are performed on a blockchain. Though not Turing complete as Ethereum [13], Bitcoin scripts can be viewed as a weak form of smart contracts, that provide important functionality, e.g. by governing the distribution of the Bitcoin cryptocurrency.

As smart contracts, including Bitcoin scripts, can control real world values and are immutable once deployed on the blockchain network, a method to demonstrate their security and correctness is needed [31, 50]. According to [4, 37, 19], there are two ways to verify their correctness: (1) by using mathematical methods like formal verification, which utilise theorem proving, model checking, and similar techniques, and (2) by employing testing. Theorem proving provides an extremely flexible verification method that can be applied to various types of systems including smart contracts. It can be done in interactive, automated, or hybrid mode.

In our approach, we use the interactive theorem prover Agda [2] for the verification of Bitcoin scripts. Agda is designed to be both an interactive theorem prover and a dependently typed programming language [40], therefore Agda allows us to define programs and reason about them in the same system. This reduces the danger of producing errors when translating programs from a programming language to a theorem prover, and allows to execute smart contracts in Agda directly. Another advantage of Agda is to have proofs that are checkable by hand. Other frameworks, such as Coq [43], use automatic proof search tools which usually do not provide proof certificates that could in principle be checked by hand. Human checkable proof certificates reduce the need to rely on a theorem prover being correct. The latter is desirable because of potential attacks that exploit errors in theorem provers, e.g. by creating a smart contract that contains a deliberate error together with a correctness proof that exploits the error in the theorem prover.¹ As a final point, there are some key distinctions between Agda and other theorem provers like Coq that suggest a different applicability of Agda. For example, Agda supports inductive-recursive types, while Coq does not [12]. Agda also has a more flexible pattern matching system than Coq, including support for copattern matching [12].

Main contributions. Our main contributions in this paper are:

- We argue that weakest preconditions are the appropriate notion to verify access control for Bitcoin scripts.
- We propose to aim for human-readable descriptions of weakest preconditions to support judging whether the security property of access control is satisfied.
- We describe two methods for achieving human-readable descriptions of weakest preconditions: a step-by-step approach, and a symbolic-execution-and-translation approach.
- We apply our proposed methodology to two standard Bitcoin scripts, providing fully formalised arguments in Agda.

¹ See the forum discussion on [48] for a well documented list of incorrect protocols with false correctness proofs. To hide backdoors using deliberately false correctness proofs is certainly conceivable.

In the following we explain our contributions in more detail. The paper introduces an operational semantics of the SCRIPT commands used in *Pay to Public Key Hash (P2PKH)* and *Pay to Multisig (P2MS)*, two standard scripts that govern the distribution of Bitcoins. We define the operational semantics as stack operations and reason about the correctness of such operations using Hoare triples utilising pre- and postconditions.

Weakest precondition for access control. Our verification focuses on the security property of *access control*. Access control is the restriction to access for a resource, which in our use case is access to cryptocurrencies like Bitcoin. We advocate that, in the context of Hoare triples, *weakest preconditions* are the appropriate notion to model access control: A (general) precondition expresses that when it is satisfied, access is granted, but there may be other ways to gain access without satisfying the precondition. The weakest precondition expresses that access is granted if and only if the condition is satisfied.

Human-readable descriptions. The weakest precondition can always be described in a direct way, for example as the set of states that after execution of the smart contract end in a state satisfying the given postcondition. However, such a description is meaningless to humans who want to convince themselves that the smart contract is secure, in the sense that they do not provide any further insights beyond the original smart contract.

It is known in software engineering, that failures of safety-critical systems are often due to incomplete requirements or specifications rather than coding errors.² The same applies to security related software.³ It is not sufficient to have a proof of security of a protocol, if the statement does not express what is required. That the specification (here the formal statement of secure access control) guarantees that the requirements are fulfilled (namely that it is impossible for a hacker to access the resource, here the Bitcoin), needs to be checked by a human being, who needs to be able to read the specification and determine whether it really is what is expressed by the requirements. Thus, the challenge is to obtain simple, human-readable descriptions of the weakest precondition of a smart contract. This would allow to close the validation gap between user requirements and formal specification of smart contracts.

Two methods for obtaining human-readable weakest preconditions. We discuss two methods for obtaining readable weakest preconditions: The first, step-by-step approach, is obtained by working through the program backwards instruction by instruction. In some cases it is easier to group several instructions together and deal with them differently, as we will demonstrate with an example in Sect. 6.3. The second method, symbolic-execution-and-translation, evaluates the program in a symbolic way, and translates it into a nested case distinction. The case distinctions are made on variables (of type *nat* or *stack*) or on expressions formed from variables by applying basic functions to them such as hashing or checking for signature. From the resulting decision tree, the weakest precondition can be read off as the disjunction of the conjunctions of the conditions that occur along branches that lead to a successful outcome.

² For instance, [32] writes: “Almost all accidents with serious consequences in which software was involved can be traced to requirements failures, and particularly to incomplete requirements.”

³ The long list of protocols which were proven to be secure but had wrong proofs [20] demonstrates that a proof of correctness is not sufficient. We assume that most of the examples had correct proofs, but the statement shown was not sufficient to guarantee security.

For both methods, it is necessary to prove that the established weakest precondition is indeed the weakest precondition for the program under consideration. For the first method, this follows by stepwise operation. The second uses a proof that the original program is equivalent to the transformed program from which the weakest precondition has been established, or a direct proof which follows the case distinctions used in the symbolic evaluation.

Application of our proposed methodology. We demonstrate the feasibility of our approaches by carrying them out in Agda for concrete smart contracts, including P2PKH and P2MS.

Our approach also provides opportunities for further applications: The usage of the weakest precondition with explicit proofs can be seen as a method of building verified smart contracts that are *correct by construction*. Instead of constructing a program and then evaluating it, one can start with the intended weakest precondition and postcondition, add some intermediate conditions, and then develop the program between those conditions. Such an approach would extend the SPARK Ada framework [1] to use Hoare logic (without the weakest precondition) to check programs.

The remainder of this paper is organised as follows. In Sect. 2, we introduce related work on verification of smart contracts. Sect. 3 introduces Bitcoin SCRIPT and defines its operational semantics. In Sect. 4, we specify the security of Bitcoin SCRIPT using Hoare logic and weakest preconditions. We formalise these notions in Agda and introduce equational reasoning for Hoare triples to streamline our correctness proofs. Sect. 5 introduces our first, step-by-step method of developing human-readable weakest preconditions and proving correctness of P2PKH. In Sect. 6, we introduce our second method based on symbolic execution and apply it to various examples. In Sect. 7, we explain how to practically use Agda to determine and prove weakest preconditions using our library [47]. We conclude in Sect. 8.

Notations and git repository. The formulas can be presented as full Agda code, but often the formulas can also be presented in mathematical style. In order to switch between Agda code and mathematical code easy, we use the functional style for application (i.e. writing $f a b c$ instead of $f(a, b, c)$) and $x : A$ instead of $x \in A$. $s :: l$ denotes prepending an element onto a list. The original Agda definitions are also available [47]. Most display style Agda code presented in this paper has been automatically extracted from the Agda code, in some cases it was formatted by hand based on L^AT_EX code generated by Agda to improve the presentation.

2 Related Work

In this section, we describe research relevant to our approach. We start by discussing two papers introducing Hoare logic, predicate transformer semantics and weakest preconditions. We then review papers that address verification of smart contracts, and Bitcoin scripts. We present a number of approaches to use model-checking for the verification of smart contracts, and finish with work with employs Agda in the verification of smart contracts.

Hoare Logic, Predicate Transformer Semantics and Weakest Preconditions. Hoare [26] defines a formal system using logical rules for reasoning about the correctness of computer programs. It uses so-called Hoare triples which combine two predicates, a pre- and a postcondition, with a program to express that if the precondition holds for a state and

the program executes successfully, then the postcondition holds for the resulting state. Dijkstra [22] introduces predicate transformer semantics that assigns to each statement in an imperative programming paradigm a corresponding total function between two predicates on the state space of the statement. The predicate transformer defined by Dijkstra applied to a postcondition returns the weakest precondition.

Verification of Smart Contracts. A number of authors have addressed the verification of smart contracts in Ethereum and similar platforms. Hirai [25] used Isabelle/HOL theorem prover to validate Ethereum Virtual Machine (EVM) bytecode by developing a formal model for EVM using the Lem language [36]. They use this model to prove invariants and safety properties of Ethereum smart contracts. Amani et al. [5] extended Hirai's EVM formalisation in Isabelle/HOL by a sound program logic at bytecode level. To this end, they stored bytecode sequences into blocks of straight-line code, creating a program logic that could reason about these sequences. Ribeiro et al. [45] developed an imperative language for a relevant subset of Solidity in the context of Ethereum, using a big-step semantics system. Additionally, they formalised smart contracts in Isabelle/HOL, extending the existing work. Their formalisation of semantics is based on Hoare logic and the weakest precondition calculus. Their main contributions are proofs of soundness and relative completeness, as well as applications of their machinery to verify some smart contracts including modelling of smart contract vulnerabilities. Bernardo et al. [9] present Mi-Cho-Coq, a Coq framework which has been used to formalise Tezos smart contracts written in the stack-based language Michelson. The framework is composed of a Michelson interpreter implemented in Coq, and the weakest precondition calculus to verify Michelson smart contracts' functional correctness. O'Connor [41] introduces Simplicity, a low-level, typed functional language, which is Turing incomplete. The goal of Simplicity is to improve on existing blockchain-based languages, like Ethereum's EVM and Bitcoin SCRIPT, while avoiding some of their issues. Simplicity is based on formal semantics and is specified in the Coq proof assistant. Bhargavan et al. [10] provided formalisations of EVM bytecode in F^* , a functional programming language designed for program verification. They defined a smart contract verification architecture that can compile Solidity contracts, and decompile EVM bytecode into F^* using their shallow embedding, in order to express and analyse smart contracts.

Verification of Bitcoin Scripts. Klomp et al. [30] proposed a symbolic verification theory, and a tool to analyse and validate Bitcoin scripts, with a particular focus on characterising the conditions under which an output script, which controls the successful transfer of Bitcoins, will succeed. Bartoletti et al. [8] presented BitML, a high-level domain-specific language for designing smart contracts in Bitcoin. They provided a compiler to convert smart contracts into Bitcoin transactions, and proved the correctness of their compiler w.r.t. a symbolic model for BitML and a computational model, which has been defined as well in [7] for Bitcoin. Setzer [46] developed models of the Bitcoin blockchain in the interactive theorem prover Agda. This work focuses on the formalisation of basic primitives in Agda as a basis for future work on verifying the protocols of cryptocurrencies and developing verified smart contracts.

Verification of Smart Contracts using Model Checking. A number of papers discuss tools for analysing and verifying smart contracts that utilise model checking. Kalra et al. [28] developed a framework called ZEUS whose aim is to support automatic formal verification of smart contracts using abstract interpretation and symbolic model checking. ZEUS starts from a high-level smart contract, and employs user assistance for capturing correctness and fairness

requirements. The contract and policy specification are then transformed into an intermediate language with well defined execution semantics. ZEUS then performs static analysis on the intermediate level and uses external SMT solvers to evaluate any verification properties discovered. A main focus of the work is on reducing efficiently the state explosion problem inherent in any model checking approach. Park et al. [42] proposed a formal verification tool for EVM bytecode based on KEVM, a complete formal semantics of EVM bytecode developed in the K-framework. To address performance challenges, they define EVM-specific abstractions and lemmas, which they then utilise to verify a number of concrete smart contracts. Mavridou et al. [33] introduce the VeriSolid framework to support the verification of Ethereum smart contracts. VeriSolid is based on earlier work (FSolidM) which allows to graphically specify Ethereum smart contracts as transitions systems, and to generate Solidity code from those specification. It uses model checking to verify smart contract models. Luu et al. [31] provided operational semantics of a subset of Ethereum bytecode called EtherLite, which forms the bases of their symbolic execution tool Oyente for analysing Ethereum smart contracts. Based on their tool they discovered a number of weaknesses in deployed smart contracts, including the DAO bug [23]. Filiâtre et al. [24] introduced the Why3 system, which allows writing imperative programs in WhyML, an ML dialect used for programming and specification. The system can add pre-, post- and intermediate conditions to it but does not make use of weakest precondition. Why3 can generate verification conditions for Hoare triple, which are checked using variously automated and interactive theorem provers. Why3 is used in SPARK Ada to verify its verification conditions.

Agda in the Verification of Blockchains. Finally, Agda features in several papers discussing verification of blockchains. Chakravarty et al. [16] introduce Extended UTXO (EUTXO), which extends Bitcoin’s UTXO model to enable more expressive forms of validation scripts. These scripts can express general state machines and reason about transaction chains: The authors introduce a new class of state machines based on Mealy machines which they call Constraint Emitting Machines (CEM). In addition to formalising CEMs using Agda proof assistant, they demonstrate its conversion to EUTXO, and give a weak bisimulation between both systems. In [14] Chakravarty et al. introduce a generalisation of the EUTXO ledger model using native tokens which they denote EUTXOma for EUTXO with multi-assets. They provide a formalisation of the multi-asset EUTXO model in Agda. Chakravarty et al. [15] introduce a version of EUTXOma aligned to Bitcoin’s UTXO model, hence denoted UTXOma. They present a formal specification of the UTXOma ledger rules and formalise their model in Agda. Chapman et al. [17] formalise System $F_{\omega\mu}$, which is polymorphic λ -calculus with higher-kinded and arbitrary recursive types, in Agda. System $F_{\omega\mu}$ corresponds to Plutus Core, which is the core of the smart contract language Plutus that features in the Cardano blockchain. Melkonian [34] introduces a formal Bitcoin transaction model to simulate transactions in the Bitcoin environment and to study their safety and correctness. The paper presents a formalisation of a process calculus for Bitcoin smart contracts, denoted BitML. The calculus can accept different types such as basic types, contracts, or small step semantics to outline a “certified compiler” [35].

3 Operational Semantics for Bitcoin Script

We give a brief introduction of Bitcoin SCRIPT in Subsec. 3.1, before defining its operational semantics in Subsec. 3.2.

3.1 Introduction to Bitcoin Script

The scripting language for Bitcoin is stack-based, inspired by the programming language Forth [44], with the stack being the only memory available. Elements on the stack are byte vectors, which we represent as natural numbers. Values on the stack are also interpreted as truth values, any value >0 will be interpreted as true, and any other value as false. SCRIPT has its own set of commands called *opcodes*, which manipulate the stack. They are similar to machine instructions, although some instructions have a more complex behaviour. The instructions of SCRIPT are executed in sequence. In case of conditionals (which are not part of this paper) the execution of instructions might be ignored until the end of an if- or else-case has been reached, otherwise the script is executed from left to right. Execution of instructions might fail, in which case the execution of the script is aborted. A full list of instructions with their meaning can be found in [11], which is the defacto specification of SCRIPT.

The operational semantics of the opcodes can be found in the source code [47]. We introduce here a number of opcodes that are relevant to this paper. Execution of all opcodes fails, if there are not sufficiently many elements on the stack to perform the operation in question.

- `OP_DUP` duplicates the top element of the stack.
- `OP_HASH` takes the top item of the stack and replaces it with its hash.
- `OP_EQUAL` pops the top two elements in the stack and checks whether they are equal or not, pushing the Boolean result on the stack.
- `OP_VERIFY` invalidates the transaction if the top stack value is false. The top item on the stack will be removed.
- `OP_CHECKSIG` pops two elements from the stack and checks whether they form a correct pair of a signature and a public key signing a serialised message obtained from the selected input and all outputs of the transaction, and pushes the Boolean result on the stack.
- `OP_CHECKLOCKTIMEVERIFY` fails if the time on the stack is greater than the current time.
- `OP_MULTISIG` is the multisig instruction, which will be discussed in detail in Sect. 6.2.
- There are a number of opcodes for pushing byte vectors of different lengths onto the stack. We write `<number>` for the opcode together with arguments pushing `number` onto the stack. In Agda we will have one instruction `opPush n` which pushes the number `n` on the stack.

Scripts can also contains control flow statements such as `OP_IF`. The verification of scripts involving control statements is more involved and will be considered in a follow-up paper.

In Bitcoin we consider the interplay between a locking script `scriptPubKey` and an unlocking script `scriptSig`.⁴ The locking script is provided by the sender of a transaction to lock the transaction, and the unlocking script is provided by the recipient to unlock it. The unlocking script pushes the data required to unlock the transaction on the stack, and the locking script then checks whether the stack contains the required data. Therefore, the unlocking script is executed first, followed by the locking script.⁵

⁴ We are using the terminology locking script and unlocking script from [6, Chapt 5].

⁵ In the original version of Bitcoin both scripts were concatenated and executed. However, because Bitcoin script has non-local instructions (e.g. the conditionals `OP_IF`, `OP_ELSE`, `OP_ENDIF`), when concatenating the two scripts any non-local opcode occurring in the locking script (for instance as part of data) could be interpreted when running as the counterpart of a non-local opcode in the locking script and therefore result in an unintended execution of the unlocking script. As a bug fix, in a later version of Bitcoin this was modified by having a break point in between the two, where only the stack is passed on. See

The main example in this paper is the pay-to-public-key-hash (P2PKH) script consisting of the following locking and unlocking scripts:

```
scriptPubKey: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUAL OP_VERIFY OP_CHECKSIG
scriptSig:    <sig> <pubKey>
```

The standard unlocking script `scriptSig` pushes a signature `sig` and a public key `pubKey` onto the stack. The locking script `scriptPubKey` checks whether `pubKey` provided by the unlocking script hashes to the provided `pubKeyHash`, and whether the signature is a signature for the message signed by the public key. Full details will be discussed in Sect. 5.

3.2 Operational Semantics

Opcodes like `OP_DUP` operate on the stack defined in Agda as a list of natural numbers `Stack`. Opcodes like `OP_CHECKSIG` check for signatures for the part of the transaction which is to be signed – what is to be signed is hard coded in Bitcoin. In order to abstract away from the precise format and the encoding, we define a message type `Msg` in Agda, which allows to represent messages such as those for the transaction to be signed, and is to be instantiated with the concrete message to be signed. Other opcodes like `OP_CHECKLOCKTIMEVERIFY` refer to the current time, for which we define a type `Time` in Agda. Therefore, the operational semantics of opcodes depends on `Time × Msg × Stack` which we define in Agda as the record type `StackState`.⁶ Note that `Time` and `Msg` don't change during the execution of a script.

The type of all opcodes is given as `InstructionBasic`.⁷ Opcodes can also fail, for instance if there are not enough elements on the stack as required by the operation. Hence, the operational semantics of an instruction $op : \text{InstructionBasic}$ is given as $\llbracket op \rrbracket_s : \text{StackState} \rightarrow \text{Maybe StackState}$.⁸

The message and time never change, so $\llbracket p \rrbracket_s s$ will, if executed successfully, only change the stack part of s . As an example, we can define the semantics of the instruction `opEqual`. We first define a simpler function $\llbracket _ \rrbracket_s^s$, which abstracts away the non-changing components `Time` and `Msg`:

$$\begin{aligned} \llbracket _ \rrbracket_s^s &: \text{InstructionBasic} \rightarrow \text{Time} \rightarrow \text{Msg} \rightarrow \text{Stack} \rightarrow \text{Maybe Stack} \\ \llbracket \text{opEqual} \rrbracket_s^s \text{ time}_1 \text{ msg} &= \text{executeStackEquality} \end{aligned}$$

The function `executeStackEquality` : `Stack` → `Maybe Stack` fails and returns `nothing` if the stack has height ≤ 1 , and otherwise compares the two top numbers on the stack, replacing them by 1 for true in case they are equal, and by 0 for false otherwise.

Chapter 6, “Separate execution of unlocking and locking scripts” in [6, p. 136]. In this paper this problem doesn't occur because we don't consider non-local instructions.

⁶ The idea of packaging all components of the state into one product type, which is then expanded into a more expanded state as more language constructs are added to the language, is inspired by Peter Mosses' Modular SOS approach [21]. This approach was successful in creating a library of reusable components funcons for defining an executable operational semantics of language constructs, which require different sets of states. One outcome was a “component-based semantics for CAML LIGHT” [18].

⁷ We are using in this paper a sublanguage `BitcoinScriptBasic` of Bitcoin, which doesn't contain conditionals, because they require a more complex operational semantics and state (see the discussion in the conclusion). We make the distinction between the basic and full language in order to be compatible with the planned follow up papers based on code under development, which will extend the basic language. We sometimes use notations such as ^b to differentiate between functions referring to the basic and full language.

⁸ For the reader not familiar with the `Maybe` type, a set theoretic notation can be given as $\text{Maybe } X := \{\text{nothing}\} \cup \{\text{just } x \mid x : X\}$. Here, `nothing` denotes undefined, and `just x` denotes the defined element x . `Maybe` forms a monad, with `return` := `just` : $A \rightarrow \text{Maybe } A$ and the bind operation $(p \gg= q : \text{Maybe } B)$ for $p : \text{Maybe } A$ and $q : A \rightarrow \text{Maybe } B$ defined by $(\text{nothing} \gg= q) = \text{nothing}$ and $(\text{just } a \gg= q) = q a$.

$\llbracket _ \rrbracket_s^5$ is then lifted to the semantics of the instructions $\llbracket _ \rrbracket_s$ using a generic function `liftStackFun2StackState`:

```

 $\llbracket \_ \rrbracket_s : \text{InstructionBasic} \rightarrow \text{StackState} \rightarrow \text{Maybe StackState}$ 
 $\llbracket op \rrbracket_s = \text{liftStackFun2StackState } \llbracket op \rrbracket_s^5$ 

```

As prerequisites for Sect 6.1, we define functions that define the operational semantics of further Bitcoin instructions used in this paper: `executeStackDup` : `Stack` \rightarrow `Maybe Stack` fails and returns `nothing` if the stack is empty; otherwise, a duplicate of the top element will be added onto the stack. The function `executeOpHash` : `Stack` \rightarrow `Maybe Stack` fails and returns `nothing` if the stack is empty; otherwise, the top element is replaced by its hash. `executeStackVerify` : `Stack` \rightarrow `Maybe Stack` fails and returns `nothing` if the stack is empty or the top element is 0; otherwise, it will remove the top element of the stack. `executeStackCheckSig` : `Stack` \rightarrow `Maybe Stack` fails and returns `nothing` if the height of the stack ≤ 1 . Otherwise it pops the two top elements from the stack, and considers them as a signature and public key. It decides whether the message given by the argument `msg` : `Msg` is correctly signed by these data, and pushes the Boolean result on the stack.

SCRIPT has instructions with more complex behaviour, an example is the instruction `OP_MULTISIG` which will be introduced in Sect. 6.2. Some instructions depend on cryptographic functions for hashing and checking signatures. We abstract away from their concrete definition and take them as parameters of the modules of the Agda code. This is not a problem in this paper, since the weakest preconditions only depend on the results returned by these functions, such as a check whether the part of the transaction to be signed is signed by a signature corresponding to a given public key.

General scripts are formalised in Agda as lists of instructions, `BitcoinScriptBasic`. Let p be a script. We define $\llbracket p \rrbracket : \text{StackState} \rightarrow \text{Maybe StackState}$ by monadic composition, that is

- $\llbracket [] \rrbracket := \text{just}$,
- for an instruction op , script q and $s : \text{StackState}$ define $\llbracket op :: q \rrbracket s := \llbracket op \rrbracket_s s \gg= \llbracket q \rrbracket$.

It follows that $\forall s : \text{StackState}. \llbracket p ++ q \rrbracket s \equiv \llbracket p \rrbracket s \gg= \llbracket q \rrbracket$.

We lift as well $\llbracket p \rrbracket$ to $s : \text{Maybe StackState}$ by defining $\llbracket p \rrbracket^+ s := s \gg= \llbracket p \rrbracket$.

Let

`StackStatePred` = `StackState` \rightarrow `Set`,

`StackPredicate` = `Time` \rightarrow `Msg` \rightarrow `Stack` \rightarrow `Set`, and

`stackPred2SPred` : `StackPredicate` \rightarrow `StackStatePred` be the obvious lifting.

4 Specifying Security of Bitcoin Scripts

In this section, we argue that weakest precondition in the context of Hoare logic are the appropriate notion to express security properties in Subsect. 4.1. We provide a formalisation of weakest preconditions in Agda in Subsect. 4.2, and discuss how weakest preconditions can be generated automatically in Subsect. 4.3, leading to the claim that we need human-readable descriptions of weakest preconditions. To support our verification, we develop a library for equational reasoning with Hoare triples in Subsect. 4.4.

4.1 Weakest Precondition for Security

One widely used way to specify the correctness of imperative programs axiomatically is Hoare logic [26]. Hoare logic is based on pre- and postconditions. It works well for safety critical systems, where the set of inputs is controlled, and the aim is to guarantee a safe result. An example of a commercial system for writing safety critical systems using Hoare logic is SPARK 2014 [1].

1:10 Verification of Bitcoin Script in Agda

However, when dealing with security aspects, in particular access control, Hoare logic in general is not sufficient. The issue is that for security it is necessary to guard against malicious entries to a program. We argue that weakest preconditions in the context of Hoare logic is an appropriate notion to specify security properties. A weakest precondition expresses that it is not only sufficient, but as well necessary for the postcondition to hold after executing the program.

To explain our point, we specify the intended correctness of the locking script `scriptPubKey` from Sect. 3. The intention, usually given by the user requirement, is that in order for a locking script to run successfully, we need to provide a public key *pbk* and a signature *sig* such that *pbk* hashes to the value `<pubKeyHash>` stored in the locking script, and that *sig* validates the signed message using *pbk*. The values *pbk* and *sig* need to be the top elements on the stack. If we also fix their order and allow the stack to have arbitrary values otherwise,⁹ then we can express this condition as follows:

The two top elements of the stack are *pbk* and *sig*, *pbk* hashes to `<pubKeyHsh>`, (CondPBKH) and *sig* is a valid signature of the signed message w.r.t. *pbk*.

We can define the specification of the locking script `scriptPubKey` as the property that (CondPBKH) is the weakest precondition for the accepting postcondition. We will show in Sect. 5 that (CondPBKH) is indeed the weakest precondition of `scriptPubKey`, which verifies that `scriptPubKey` fulfils the specification.

Let us now consider a faulty locking script instead of `scriptPubKey`:

```
scriptPubKeyFaulty: OP_DUP OP_HASH160 <pubKeyHash> OP_EQUAL
```

To see that it does not fulfil the specification given above, consider the weakest precondition for `scriptPubKeyFaulty` for the accepting postcondition, which can be described by the following condition:

The top element of the stack is *pbk*, and *pbk* hashes to `<pubKeyHsh>`. (CondPBKHfaulty)

By inspection we see that (CondPBKHfaulty) is not equivalent to (CondPBKH), and therefore `scriptPubKeyFaulty` doesn't fulfil the specification. In fact we can identify states which satisfy (CondPBKHfaulty) but not (CondPBKH), e.g. a malicious attacker could just copy the public key of the sender onto the stack, which violates the user requirements of a locking script.

We observe that this example also demonstrates the inadequacy of general Hoare logic for the verification the security property of access control: Using standard Hoare logic, we can prove that (CondPBKH) is a precondition for the accepting postcondition for both `scriptPubKey` and `scriptPubKeyFaulty`.

As with all formal verification approaches, there remains a gap between the user's intention expressed as requirements, and what is expressed as a formal specification. This gap cannot be filled in a provably correct way, since requirements are a mental intention expressed in natural language. However, the gap can be narrowed by expressing the specification in a human-readable format so that the validation is as easy and clear as possible. Here, validation means showing that the specification guarantees the requirements, and is carried out by a human reader.

⁹ Bitcoin scripts do not put any requirements on the stack below the data required by the scripts.

4.2 Formalising Weakest Preconditions in Agda

We now describe how weakest preconditions can be defined in Agda. Let a precondition φ and postcondition ψ be given, both of type `StackStatePred`. In order to accommodate `Maybe`, we define a postfix operator $_+$, to lift ψ to (ψ^+) : `Maybe StackState` \rightarrow `Set`, defining $(\psi^+) \text{ nothing} = \perp$ and $(\psi^+) \circ \text{just} = \psi$.

A Hoare triple, consisting of a precondition, a program, and a postcondition, expresses that if the precondition is satisfied before execution of the program, then the postcondition holds after executing it. We formalise Hoare triples as follows:

$$\langle \varphi \rangle p \langle \psi \rangle := \forall s \in \text{StackState}. \varphi(s) \rightarrow (\psi^+) (\llbracket p \rrbracket s)$$

Weakest preconditions express that the precondition not only is sufficient, but as well necessary for the postcondition to hold after executing the program:

$$\langle \varphi \rangle \text{iff } p \langle \psi \rangle := \forall s \in \text{StackState}. \varphi(s) \leftrightarrow (\psi^+) (\llbracket p \rrbracket s)$$

Thus, for security the backwards direction of the equivalence in the previous formula is the important direction.

In Bitcoin we consider a locking script `scriptPubKey` and an unlocking script `scriptSig`, see Section 3.1. Let us fix an unlocking script `unlock` and a locking script `lock`. Let `init` be the initial state consisting of an empty stack, and let `acceptState` be the accepting condition expressing that the stack is non empty with top element being not false, i.e. >0 . The combination of `unlock` and `lock` is accepted iff running `unlock` on `init` succeeds and running `lock` on the resulting stack results in a state that satisfies the accepting condition, i.e. iff $(\text{acceptState}^+) (\llbracket \text{lock} \rrbracket^+ (\llbracket \text{unlock} \rrbracket \text{init}))$. Note that Bitcoin does not run the concatenation of the two scripts, as it did in its first version, but runs first the unlocking scripts, and if it succeeds runs the locking script on the resulting stack. Let φ be the weakest precondition of `lock`, i.e. $\langle \varphi \rangle \text{iff } \text{lock} \langle \text{acceptState} \rangle$. Then the acceptance condition is equivalent to $(\varphi^+) (\llbracket \text{unlock} \rrbracket \text{init})$. Thus, `unlock` succeeds iff running the unlocking script `unlock` on the initial state `init` produces a state fulfilling φ . Hence, by determining the weakest precondition for the locking script w.r.t. the accepting condition we have obtained a characterisation of the set of unlocking scripts which unlock the locking script. Note that we do not define inductively all successful unlocking scripts, since they could be arbitrary complex programs, but instead characterise them by the output they produce.

4.3 Automatically Generated Weakest Preconditions

We start by giving a direct method for defining the weakest precondition for any Bitcoin script by describing the set of states that lead to a given final state. We then apply this general method to a toy example to demonstrate that the description obtained in this way is usually not helpful for a human to judge whether the script has the right properties, thus making the case that the task must be to find (equivalent) human-readable descriptions.

Weakest preconditions can be defined by the simple definition

$$\begin{aligned} \text{weakestPreCond}^s &: \text{BitcoinScriptBasic} \rightarrow \text{StackStatePred} \rightarrow \text{StackStatePred} \\ \text{weakestPreCond}^s p \phi s &= (\phi^+) (\llbracket p \rrbracket s) \end{aligned}$$

Consider a simple toy program which removes the top element from the stack three times:
`testprog = opDrop :: opDrop :: [opDrop]`

Its weakest precondition can be computed as

$$\text{weakestPreCondTestProg} = \text{weakestPreCond}^s \text{testprog } \text{acceptState}$$

We obtain the following code (we slightly reformatted it to improve readability):

1:12 Verification of Bitcoin Script in Agda

```

weakestPreCondTestProgNormalised s =
  (stackPred2SPred acceptStates +)
  (stackState2WithMaybe ⟨ currentTime s , msg s , executeStackDrop (stack s) ⟩
  >>= (λ s1 → stackState2WithMaybe ⟨ currentTime s1 , msg s1 , executeStackDrop (stack s1) ⟩
      >>= liftStackFun2StackState (λ time1 msg1 → executeStackDrop)))

```

This condition is difficult to understand. The reason is that each instruction may cause the program to abort in case the stack is empty. The condition expresses: if the stack is empty then the condition is false. Otherwise, if after dropping the top element the stack is empty the condition is false. Otherwise, if after dropping again the top element the stack is empty the condition is false. Otherwise the condition is true if after dropping again the top element the stack is non empty and the top element is not false. The readable condition would express that the height of the stack is ≥ 4 and the fourth element from the top is > 0 . In this simple example simplifying the condition would be easy, but when using different instructions the situation becomes more complicated.

What we did using our methods to avoid this problem was to create the weakest precondition by starting from the end and improving it in each step, or by replacing the program by an easier program (which in case of this example would return nothing if the stack has height ≤ 2 and otherwise returns the result of dropping the first three elements off the stack). An interesting project for future work would be to automate the steps we carried out manually, and obtain readable weakest preconditions automatically.

4.4 Equational Reasoning with Hoare Triples

To support the verification of Bitcoin scripts with Hoare triples and weakest preconditions in Agda, we have developed a library in Agda for equational reasoning with Hoare triples. The library is inspired by what is described in Wadler et al. [49]. Let p, q be scripts and $\phi, \phi', \psi, \psi' : \text{Predicate}$. If we define $\phi \langle \Rightarrow \rangle^P \psi := \forall s : \text{StackState}. \phi(s) \leftrightarrow \psi(s)$, we can easily show

$$\begin{aligned}
\langle \phi \rangle^{\text{iff}} p \langle \psi \rangle \wedge \langle \psi \rangle^{\text{iff}} q \langle \rho \rangle &\rightarrow \langle \phi \rangle^{\text{iff}} p ++ q \langle \rho \rangle \\
\langle \phi \rangle^{\text{iff}} p \langle \psi \rangle \wedge \psi \langle \Rightarrow \rangle^P \psi' &\rightarrow \langle \phi \rangle^{\text{iff}} p \langle \psi' \rangle \\
\phi' \langle \Rightarrow \rangle^P \phi \wedge \langle \phi \rangle^{\text{iff}} p \langle \psi \rangle &\rightarrow \langle \phi' \rangle^{\text{iff}} p \langle \psi \rangle
\end{aligned}$$

We demonstrate our syntax by an example, assuming (using Agda postulate) programs `prog1`, `prog2`, `prog3`, and proofs

```

proof1 : ⟨ precondition ⟩iff prog1 ⟨ intermediateCond1 ⟩
proof2 : ⟨ intermediateCond1 ⟩iff prog2 ⟨ intermediateCond2 ⟩
proof3 : intermediateCond2 <=>P intermediateCond3
proof4 : ⟨ intermediateCond3 ⟩iff prog3 ⟨ postcondition ⟩

```

Then the proof for the Hoare triple for `prog1 ++ (prog2 ++ prog3)` is given in Agda as follows:¹⁰

```

theorem : ⟨ precondition ⟩iff prog1 ++ (prog2 ++ prog3) ⟨ postcondition ⟩
theorem = precondition <><>⟨ prog1 ⟩⟨ proof1 ⟩

```

¹⁰In the last step we use `>e` instead of `>`. This avoids concatenating the program with `[]`. If we used `>`, the theorem would prove the condition for program `prog1++(prog2++(prog3++[]))`, which is provably but not definitionally equal to the original program, requiring an additional proof step.

```

intermediateCond1 <><>< prog2 \> proof2 )
intermediateCond2 <=>< proof3 )
intermediateCond3 <><>< prog3 \> proof4 )e postcondition ■p

```

5 Proof of Correctness of the P2PKH script using the Step-by-Step Approach

This section explains the usage of our approach by providing an example of how to prove the correctness of the P2PKH using step-by-step to obtain the weakest precondition. The P2PKH is the most used script in Bitcoin transactions. The locking script, which depends on a public key hash, is defined as follows:

```

scriptP2PKHb : (pbkh : ℕ) → BitcoinScriptBasic
scriptP2PKHb pbkh = opDup :: opHash :: (opPush pbkh) :: opEqual :: opVerify :: [ opCheckSig ]

```

In this section, we develop a readable weakest precondition of the P2PKH script and prove its correctness by working backwards instruction by instruction.

Let `acceptState` be the accepting state where the stack is non-empty with top element >0 . We define intermediate conditions `accept1`, `accept2`, etc, the weakest precondition `wPreCondP2PKH`, and proofs `correct-opCheckSig`, `correct-opVerify` etc of corresponding Hoare triples w.r.t. the instructions of the Bitcoin script, working backwards starting from the last instruction `opCheckSig`:

```

correct-opCheckSig : < accept1 >iff ([ opCheckSig ] ) < acceptState >
correct-opVerify : < accept2 >iff ([ opVerify ] ) < accept1 >
correct-opEqual : < accept3 >iff ([ opEqual ] ) < accept2 >
correct-opPush : (pbkh : ℕ) → < accept4 pbkh >iff ([ opPush pbkh ] ) < accept3 >
correct-opHash : (pbkh : ℕ) → < accept5 pbkh >iff ([ opHash ] ) < accept4 pbkh >
correct-opDup : (pbkh : ℕ) → < wPreCondP2PKH pbkh >iff ([ opDup ] ) < accept5 pbkh >

```

The intermediate conditions can be read off from the operations. We present them in mathematical notation below, using the following conventions and abbreviations: $t : \mathbb{N}$ denotes time, $m : \text{Msg}$, $st, st' : \text{Stack}$, $x : \mathbb{N}$; for brevity we omit types after \exists quantifiers. We use here and in the remaining paper ⁵ for operations where the `StackState` argument has been unfolded into its components.

$$\begin{aligned}
\text{acceptState}^s t m st &\Leftrightarrow \exists x, st'. && st \equiv x :: st' \wedge x > 0 \\
\text{accept}_1^s t m st &\Leftrightarrow \exists pbk, sig, st'. && st \equiv pbk :: sig :: st' \\
&&& \wedge \text{IsSigned } m \text{ sig } pbk \\
\text{accept}_2^s t m st &\Leftrightarrow \exists x, pbk, sig, st'. && st \equiv x :: pbk :: sig :: st' \\
&&& \wedge x > 0 \wedge \text{IsSigned } m \text{ sig } pbk \\
\text{accept}_3^s t m st &\Leftrightarrow \exists pbkh_2, pbkh_1, pbk, sig, st'. && st \equiv pbkh_2 :: pbkh_1 :: pbk :: sig :: st' \\
&&& \wedge pbkh_2 \equiv pbkh_1 \wedge \text{IsSigned } m \text{ sig } pbk \\
\text{accept}_4^s pbkh_1 t m st &\Leftrightarrow \exists pbkh_2, pbk, sig, st'. && st \equiv pbkh_2 :: pbk :: sig :: st' \\
&&& \wedge pbkh_2 \equiv pbkh_1 \wedge \text{IsSigned } m \text{ sig } pbk \\
\text{accept}_5^s pbkh_1 t m st &\Leftrightarrow \exists pbk_1, pbk, sig, st'. && st \equiv pbk_1 :: pbk :: sig :: st' \\
&&& \wedge \text{hashFun } pbk_1 \equiv pbkh_1 \wedge \text{IsSigned } m \text{ sig } pbk \\
\text{wPreCondP2PKH}^s pbkh_1 t m st &\Leftrightarrow \exists pbk, sig, st'. && st \equiv pbk :: sig :: st' \\
&&& \wedge \text{hashFun } pbk \equiv pbkh_1 \wedge \text{IsSigned } m \text{ sig } pbk
\end{aligned}$$

1:14 Verification of Bitcoin Script in Agda

In Agda, these formulas are defined by case distinction on the stack. As examples, the code for the accept condition (`acceptState`) and the weakest precondition (`wPreCondP2PKHs`) is as follows:

```

acceptStates : StackPredicate
acceptStates time msg1 [] = ⊥
acceptStates time msg1 (x :: stack1) = NotFalse x

wPreCondP2PKHs : (pbkh : ℕ) → StackPredicate
wPreCondP2PKHs pbkh time m [] = ⊥
wPreCondP2PKHs pbkh time m (x :: []) = ⊥
wPreCondP2PKHs pbkh time m (pbk :: sig :: st) =
  (hashFun pbk ≡ pbkh) ∧ IsSigned m sig pbk

```

Using our syntax for equational reasoning, we can prove the weakest precondition for the P2PKH script as follows:

```

theoremP2PKH : (pbkh : ℕ) → < wPreCondP2PKH pbkh >iff scriptP2PKHb pbkh < acceptState >
theoremP2PKH pbkh = wPreCondP2PKH pbkh <><>< [ opDup ] >>< correct-opDup pbkh >
  accept5 pbkh <><>< [ opHash ] >>< correct-opHash pbkh >
  accept4 pbkh <><>< [ opPush pbkh ] >>< correct-opPush pbkh >
  accept3 <><>< [ opEqual ] >>< correct-opEqual >
  accept2 <><>< [ opVerify ] >>< correct-opVerify >
  accept1 <><>< [ opCheckSig ] >>< correct-opCheckSig >e
  acceptState ■p

```

The locking script will be accepted if, after executing the code starting with the stack returned by the unlocking script, the accept condition `acceptState` is fulfilled. The verification conditions and proofs were developed by working backwards starting from the last instruction and determining the weakest preconditions “`accepti`” w.r.t. the end piece of the script starting with that instruction and the accept condition as post-condition. The preconditions were obtained manually – one could automate this by determining for each instruction depending on the post-condition a corresponding pre-condition, where the challenge would be to simplify the resulting pre-conditions in order to avoid a blowup in size. We continued in this way until we reached the first instruction and obtained the weakest precondition for the locking script. `theoremP2PKH` is using single instructions in order to prove the correctness of P2PKH. The proofs `correct-opCheckSig`, `correct-opVerify`, etc are done by following the case distinctions made in the corresponding verification conditions. The harder direction is to prove that they are actually *weakest* preconditions: Proving that the precondition implies the postcondition after running the program, is easier since we are used to mentally executing programs in forward direction. Proving the opposite direction requires showing that the only way, after running the program, to obtain the postcondition is to have the precondition fulfilled, which requires mentally reversing the execution of programs.

6 Proof of Correctness using Symbolic Execution

In this section, we will introduce a second method for obtaining readable representations of weakest preconditions of Bitcoin scripts. This method is based on symbolic execution [29] of the Bitcoin script, and investigating the sequence of case distinctions carried out during

the execution. We will consider three examples: The first will be the P2PKH script which we analysed already. We use it to explain the method and provide a second approach to determine and verify the already obtained weakest precondition. The second example will consider the multisig script which is a direct application of the `OP_MULTISIG` instruction. The third example will see an application of a combination of both methods.

6.1 Example: P2PKH Script

When applying the symbolic evaluation method to the P2PKH script and analysing the sequence of case distinctions carried out, we will see that there will be exactly one path through the tree of case distinctions which results in an accepting condition. The conjunction of the cases that determine this path will form the weakest precondition. In examples with more than one accepting path we would take the disjunction of the conditions for each accepting path.¹¹ We will prove that the precondition is indeed the weakest by developing an equivalent program `p2pkhFunctionDecoded` and showing that it fulfils the weakest precondition.

We start by declaring (using Agda's `postulate`) symbolic values `pbkh`, `msg1`, `stack1`, `x1`, etc for the parameters (postulates are typeset in blue). This allows us to evaluate expressions up to `executeStackVerify` symbolically by using the normalisation procedure of Agda and to determine the function `p2pkhFunctionDecoded`. (In Sect. 7 we will elaborate how to do this practically in Agda). Afterwards, we stop using those postulates (they were defined as `private`) and prove that the result of evaluating the P2PKH script for arbitrary parameters is equivalent to `p2pkhFunctionDecoded`.

When evaluating $\llbracket \text{scriptP2PKH}^b \text{ pbkh} \rrbracket^s \text{ time}_1 \text{ msg}_1 \text{ stack}_1$ we obtain

```
executeStackDup stack1                >>=
λ stack2 → executeOpHash stack2      >>=
λ stack3 → executeStackEquality (pbkh :: stack3) >>=
λ stack4 → executeStackVerify stack4    >>=
λ stack5 → executeStackCheckSig msg1 stack5
```

We can write it equivalently using the `do` notation¹²

```
do stack2 ← executeStackDup stack1
   stack3 ← executeOpHash stack2
   stack4 ← executeStackEquality (pbkh :: stack3)
   stack5 ← executeStackVerify stack4
   executeStackCheckSig msg1 stack5
```

At this point further reduction is blocked by the first line of the previous expression, because `executeStackDup stack1` makes a case distinction on `stack1`. Therefore, we introduce a symbolic case distinction on `stack1`:

¹¹In our examples we got only a few accepting paths, since concrete scripts in use are designed to deal with a small number of different scenarios for unlocking them, so the majority of paths in the program are unsuccessful paths. It could happen however that with more advanced examples nested conditions result in an exponential blowup of the number of cases – if that occurs one would need to take an approach where the nested case distinctions are preserved at least partly and the resulting extracted formulas reflect those nested case distinctions rather than flattening them out. This would avoid the blowup in the size of the resulting weakest precondition.

¹²The `do` notation is a widely used Haskell notation adapted to Agda, which provides an alternative syntax for the same expression making it appear as an imperative program if one reads `←` as assignments. It demonstrates that we are consecutively executing the instructions, with the possibility of aborting in each step.

1:16 Verification of Bitcoin Script in Agda

- $\llbracket \text{scriptP2PKH}^b \text{ pbkh} \rrbracket^s \text{ time}_1 \text{ msg}_1 \llbracket \rrbracket$ evaluates to **nothing**.
- $\llbracket \text{scriptP2PKH}^b \text{ pbkh} \rrbracket^s \text{ time}_1 \text{ msg}_1 (\text{pbk} :: \text{stack}_1)$ evaluates to what in do notation can be written as

```
do stack₅ ← executeStackVerify (compareNaturals pbkh (hashFun pbk) :: pbk :: stack₁)
   executeStackCheckSig msg₁ stack₅
```

Evaluation of the latter expression is blocked by the function `executeStackVerify` which makes a case distinction on the expression `compareNaturals pbkh (hashFun pbk)`. We define

```
abstrFun : (stack₁ : Stack)(cmp : ℕ) → Maybe Stack
abstrFun stack₁ cmp = do stack₅ ← executeStackVerify (cmp :: pbk :: stack₁)
   executeStackCheckSig msg₁ stack₅
```

hence $\llbracket \text{scriptP2PKH}^b \text{ pbkh} \rrbracket^s \text{ time}_1 \text{ msg}_1 (\text{pbk} :: \text{stack}_1)$ evaluates to `abstrFun stack₁ (compareNaturals pbkh (hashFun pbk))`.

Next we carry out a symbolic case distinction on the argument `cmp` of `abstrFun`:

- `abstrFun stack₁ 0` evaluates to **nothing**.
- `abstrFun stack₁ (suc x₁)` evaluates to `executeStackCheckSig msg₁ (pbk :: stack₁)`.

In order to normalise further, `executeStackCheckSig` needs to make a case distinction on `stack₁`, so we carry out a symbolic case distinction on that argument:

- `abstrFun $\llbracket \rrbracket$ (suc x₁)` evaluates to **nothing**.
- `abstrFun (sig₁ :: stack₁) (suc x₁)` evaluates to `just (boolToNat (isSigned msg₁ sig₁ pbk) :: stack₁)`

We can now read off the weakest precondition. The only path which ends up in a **just** result is when the stack is non empty of the form `pbk :: stack₁`, and `compareNaturals pbkh (hashFun pbk)` evaluates to `suc x₁`, i.e. it must be >0 . Furthermore, in this case `stack₁` needs to be itself non empty. For `stack₁ = sig₁ :: stack₂`, the result returned is `just (boolToNat (isSigned msg₁ sig₁ pbk) :: stack₁)`, which fulfils the accept condition if `boolToNat (isSigned msg₁ sig₁ pbk) > 0`. The latter is the case if `isSigned msg₁ sig₁ pbk` is true.

Furthermore, `compareNaturals n m` returns 1 if n, m are equal otherwise 0, so it is >0 if $n = m$. Therefore the P2PKH locking script succeeds with an output stack fulfilling the acceptance condition, if and only if the input stack has height at least two, and if it is `pbk :: sig₁ :: stack₂`, then `pbkh` is equal to `hashFun pbk`, and `isSigned msg₁ sig₁ pbk` is true. That is the same as the weakest precondition that we determined using the first approach.

In order to prove correctness, we first determine a more Agda style formulation of the result of evaluation of the P2PKH script, which we derive from the previous symbolic evaluation:

```
p2pkhFunctionDecoded : (pbkh : ℕ)(msg₁ : Msg)(stack₁ : Stack) → Maybe Stack
p2pkhFunctionDecoded pbkh msg₁  $\llbracket \rrbracket$  = nothing
p2pkhFunctionDecoded pbkh msg₁ (pbk :: stack₁) = p2pkhFunctionDecodedAux1 pbk msg₁ stack₁
   (compareNaturals pbkh (hashFun pbk))
```

```
p2pkhFunctionDecodedAux1 : (pbk : ℕ)(msg₁ : Msg)(stack₁ : Stack)(cpRes : ℕ) → Maybe Stack
p2pkhFunctionDecodedAux1 pbk msg₁  $\llbracket \rrbracket$  cpRes = nothing
p2pkhFunctionDecodedAux1 pbk msg₁ (sig₁ :: stack₁) zero = nothing
p2pkhFunctionDecodedAux1 pbk msg₁ (sig₁ :: stack₁) (suc cpRes) =
   just (boolToNat (isSigned msg₁ sig₁ pbk) :: stack₁)
```


We prove that this function is equivalent to the result of evaluating the P2PKH script. The proof is a simple case distinction following the cases defining `p2pkhFunctionDecoded`:

$$\begin{aligned} \text{p2pkhFunctionDecodedcor} &: (time_1 : \mathbb{N}) (pbkh : \mathbb{N})(msg_1 : \text{Msg})(stack_1 : \text{Stack}) \\ &\rightarrow \llbracket \text{scriptP2PKH}^b \text{ pbkh} \rrbracket^s time_1 msg_1 stack_1 \equiv \text{p2pkhFunctionDecoded pbkh msg}_1 stack_1 \end{aligned}$$

We show that the extracted weakest precondition is a correct for the extracted program:¹³

$$\begin{aligned} \text{lemmaPTKHcoraux} &: (pbkh : \mathbb{N}) \rightarrow \langle \text{weakestPreConditionP2PKH}^s \text{ pbkh} \rangle^g \\ &\quad (\lambda time msg_1 s \rightarrow \text{p2pkhFunctionDecoded pbkh msg}_1 s) \\ &\quad \langle \text{acceptState}^s \rangle \end{aligned}$$

Afterwards, this is transferred into a proof of the weakest precondition for the P2PKH script, using the equality proof from before:

$$\text{theoPTPKHcor} : (pbkh : \mathbb{N}) \rightarrow \langle \text{wPreCondP2PKH pbkh} \rangle^{\text{iff}} \text{scriptP2PKH}^b \text{ pbkh} \langle \text{acceptState} \rangle$$

Carrying out the symbolic execution was relatively easy, because Agda supports evaluation of terms very well. It only becomes relatively long in the Agda code [47] when documenting all the steps, which we did in order to explain how this is done in detail. What matters is the resulting program and a prove that it is equivalent, which was relatively short and easy. Maybe Agda's reflection mechanism [3], once it is more fully developed, could be of help to find the successful branches of the program more easily. To obtain a readable program rather than a machine generated program, and therefore readable verification conditions, would however require a lot of work, and probably require delegating some programming tasks from Agda (in which tactics need to be written) to its foreign language interface.

6.2 Example: MultiSig Script (P2MS)

The `OP_MULTISIG` instruction is an instruction which has a more complex behaviour: It assumes that the top elements of the stack are as follows:

$$n :: pbk_n :: \dots :: pbk_2 :: pbk_1 :: m :: sig_m :: \dots :: sig_2 :: sig_1 :: dummy$$

`OP_MULTISIG` checks whether the m signatures are signatures corresponding to m of the n public keys for the `msg` to be signed, where the matching public keys are in the same order as the signatures. Observe that when pushed from a script, the public keys and signatures appear in reverse order on the stack, as `pbk1` is pushed first onto the stack, etc. The `dummy` element occurs due to a mistake in the Bitcoin protocol, which has not been corrected as it would require a hard fork [6, p. 151-152].

The operational semantics is given by a function `executeMultiSig`, which fetches the data from the stack as described before. It fails if there are not enough elements on the stack and otherwise returns `just (boolToNat (cmpMultiSigs msg sigs pbks) :: restStack)`, where `sigs` and `pbks` are the signatures and public keys fetched from the stack in reverse order, and `restStack` is the remainder of the stack. The function `cmpMultiSigs` compares whether signatures correspond to public keys and is defined as follows:

¹³ $\langle _ \rangle^g \langle _ \rangle$ is the generalisation of $\langle _ \rangle^{\text{iff}} \langle _ \rangle$ where Bitcoin scripts are replaced by Agda functions `StackState` \rightarrow `Maybe StackState`; $\langle _ \rangle^g \langle _ \rangle$ is the version, where the `StackState` is unfolded into its components.

1:18 Verification of Bitcoin Script in Agda

```

cmpMultiSigs : (msg : Msg)(sigs pbks : List ℕ) → Bool
cmpMultiSigs msg [] pubkeys           = true
cmpMultiSigs msg (sig :: sigs) []     = false
cmpMultiSigs msg (sig :: sigs) (pbk :: pbks) = cmpMultiSigsAux msg sigs pbks sig (isSigned msg sig pbk)

cmpMultiSigsAux : (msg : Msg)(sigs pbks : List ℕ)(sig : ℕ)(testRes : Bool) → Bool
cmpMultiSigsAux msg sigs pbks sig false = cmpMultiSigs msg (sig :: sigs) pbks
cmpMultiSigsAux msg sigs pbks sig true  = cmpMultiSigs msg sigs pbks

```

We define now a generic multisig function. First we define `opPushList`, which pushes a list of public keys on the stack:

```

opPushList : (pbkList : List ℕ) → BitcoinScriptBasic
opPushList [] = []
opPushList (pbk1 :: pbkList) = opPush pbk1 :: opPushList pbkList

```

The m out of n multi-signature script P2MS ($n = \text{length } pbkList$) is defined as follows:

```

multiSigScript $m$ - $n$ b : (m : ℕ)(pbkList : List ℕ)( $m < n$  : m < length pbkList)
  → BitcoinScriptBasic
multiSigScript $m$ - $n$ b m pbkList  $m < n$  =
  opPush m :: (opPushList pbkList ++ (opPush (length pbkList) :: [ opMultiSig ]))

```

The locking script MultiSig script P2MS applies `OP_MULTISIG` to m signatures and n public keys. It pushes the number m of required signatures, then n public keys, and then the number n as the number of public keys, onto the stack, and executes `OP_MULTISIG`. If `OP_MULTISIG` finds that the m signatures are valid signature for the message to be signed for m out of the n public keys in the same order as they appear in the list of public keys, then the script will be unlocked. As unlocking script one can use `opPushList` applied to a list of m appropriate signatures. In order to verify the script we will consider the concrete example of the 2-out-of-4 P2MS, for which we obtain a very readable verification condition (the generic one becomes difficult to read).

We will use the second approach of determining a readable form of the weakest precondition and proving correctness by symbolic evaluation for the 2 out of 4 `multiSigScript2-4b`. The first approach is difficult to carry out since the instruction `opMultiSig` has a very complex precondition that is difficult to handle – it requires that the stack contains the number of public keys, then the public keys themselves, then the number of signatures and the signatures, and a dummy element, where the number of public keys and number of signatures can be arbitrary. It is much easier to handle the full `multiSigScript2-4b` script, since, after the data has been inputted, the number of required signatures is known, and the public keys are already provided by the script.

In order to demonstrate the first approach we will instead, in Subsect. 6.3, apply the step-by-step approach to a combined script, of which `multiSigScript2-4b` is one part. This way we obtain a readable form of the weakest precondition and can then prove its correctness. This will demonstrate that in some cases it is beneficial to interleave the two processes, and apply the second method to sequences of instructions while applying the first approach to the resulting sequences of instructions instead of single instructions.

We start the symbolic evaluation by computing the normal form of

```

[[ multiSigScript2-4b pbk1 pbk2 pbk3 pbk4 ]]s time1 msg1 stack1

```

and obtain

```

executeMultiSig3 msg1 (pbk1 :: pbk2 :: pbk3 :: [ pbk4 ]) 2 stack1 []

```

Here, `executeMultiSig3` is one of the auxiliary functions in the definition of `executeMultiSig`.

That expression makes a case distinction on `stack1` and returns:

- `nothing` when the stack has height at most 2 (obtained by evaluating it symbolically for stacks of height 0, 1, 2).
- Otherwise, the stack has height ≥ 3 , and, if it is of the form `sig2 :: sig1 :: dummy :: stack1`, it reduces to

```
just (boolToNat (cmpMultiSigsAux msg1 [ sig2 ] (pbk2 :: pbk3 :: [ pbk4 ])) sig1
      (isSigned msg1 sig1 pbk1)) :: stack1)
```

The script has terminated, because we obtain `just` as a result of the evaluation. We now need to check whether the result fulfils the accept condition. For this the top element of the stack needs to be >0 , which is the case if

`cmpMultiSigsAux msg1 [sig2] (pbk2 :: pbk3 :: [pbk4])` `sig1` `(isSigned msg1 sig1 pbk1)`

returns `true`. Therefore, we perform symbolic case distinctions in the following way:

- In case `isSigned msg1 sig1 pbk1` evaluates to `true`, i.e. if we replace that expression by `true`, the reduction continues to `cmpMultiSigsAux msg1 [] (pbk3 :: [pbk4])` `sig2` `(isSigned msg1 sig2 pbk2)`, which makes a case distinction on `isSigned msg1 sig2 pbk2`.
 - If that expression returns again `true`, we obtain `true`.
 - If it returns false, we obtain `cmpMultiSigsAux msg1 [] [pbk4]` `sig2` `(isSigned msg1 sig2 pbk3)` which makes a case distinction on `isSigned msg1 sig2 pbk3`
 - * In case of `true`, we obtain `true`.
 - * Otherwise the case distinctions continue, see the git repository [47] for full details.

In total we see that we obtain `true` iff one of the following cases holds:

- `(isSigned msg1 sig1 pbk1) ∧ (isSigned msg1 sig2 pbk2)`
- `(isSigned msg1 sig1 pbk1) ∧ ¬ (isSigned msg1 sig2 pbk2) ∧ (isSigned msg1 sig2 pbk3)`
- `(isSigned msg1 sig1 pbk1) ∧ ¬ (isSigned msg1 sig2 pbk2) ∧ ¬ (isSigned msg1 sig2 pbk3) ∧ (isSigned msg1 sig2 pbk4)`
- ... more cases.

These cases can be simplified to an equivalent disjunction of the following cases:

- `(isSigned msg1 sig1 pbk1) ∧ (isSigned msg1 sig2 pbk2)`
- `(isSigned msg1 sig1 pbk1) ∧ (isSigned msg1 sig2 pbk3)`
- `(isSigned msg1 sig1 pbk1) ∧ (isSigned msg1 sig2 pbk4)`
- ... more cases.

We obtain the following weakest precondition as a stack predicate:

```
weakestPreCondMultiSig-2-4s : (pbk1 pbk2 pbk3 pbk4 : ℕ) → StackPredicate
weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4 time msg1 [] = ⊥
weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4 time msg1 (x :: []) = ⊥
weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4 time msg1 (x :: y :: []) = ⊥
weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4 time msg1 ( sig2 :: sig1 :: dummy :: stack1) =
  ( (IsSigned msg1 sig1 pbk1 ∧ IsSigned msg1 sig2 pbk2) ⊕
    (IsSigned msg1 sig1 pbk1 ∧ IsSigned msg1 sig2 pbk3) ⊕
    (IsSigned msg1 sig1 pbk1 ∧ IsSigned msg1 sig2 pbk4) ⊕
    (IsSigned msg1 sig1 pbk2 ∧ IsSigned msg1 sig2 pbk3) ⊕
    (IsSigned msg1 sig1 pbk2 ∧ IsSigned msg1 sig2 pbk4) ⊕
    (IsSigned msg1 sig1 pbk3 ∧ IsSigned msg1 sig2 pbk4))
```

It expresses that the stack must have height at least 3, and if it is of the form $\text{sig}_2 :: \text{sig}_1 :: \text{dummy} :: \text{stack}_1$ then the signatures need to correspond to 2 out of the 4 public keys in the same order as the public keys. Using the same case distinctions as they occurred in the symbolic evaluation above we can now prove:

$$\begin{aligned} & \text{theoremCorrectnessMultiSig-2-4} : (pbk1\ pbk2\ pbk3\ pbk4 : \mathbb{N}) \\ & \rightarrow \langle \text{stackPred2SPred} (\text{weakestPreCondMultiSig-2-4}^s\ pbk1\ pbk2\ pbk3\ pbk4) \rangle^{iff} \\ & \quad \text{multiSigScript2-4}^b\ pbk1\ pbk2\ pbk3\ pbk4 \\ & \quad \langle \text{stackPred2SPred}\ \text{acceptState}^s \rangle \end{aligned}$$

From the theorem above, we have obtained a readable weakest precondition by symbolic execution, which will be used as a starting template for developing a generic verification. The next step would be to generalise the verification conditions and theorems to the generic case, however that would go beyond the scope of the current paper.

6.3 Example: Combining the two Methods

In this subsection, we show how to verify a combined script which consists of a simple script checking a certain amount of time has passed and the multisig script from the previous subsection. To determine a readable form of the weakest precondition and proving correctness we will combine both of our techniques: The weakest precondition for the multisig script has been determined by symbolic evaluation in the previous subsection. The weakest precondition for the simple time checking script will be obtained directly, as it is very simple. When we consider the combined scripts we will use the first method of moving backwards step-by-step. However, instead of using single instructions in each step, we now use several instructions as a single step.

We define the checktime script as follows:

$$\begin{aligned} & \text{checkTimeScript}^b : (time_1 : \text{Time}) \rightarrow \text{BitcoinScriptBasic} \\ & \text{checkTimeScript}^b\ time_1 = (\text{opPush}\ time_1) :: \text{opCHECKLOCKTIMEVERIFY} :: [\text{opDrop}] \end{aligned}$$

If we define

$$\begin{aligned} & \text{timeCheckPreCond} : (time_1 : \text{Time}) \rightarrow \text{StackPredicate} \\ & \text{timeCheckPreCond}\ time_1\ time_2\ msg\ stack_1 = time_1 \leq time_2 \end{aligned}$$

we can define its weakest precondition relative to a post condition ϕ only affecting the stack as in the following theorem:

$$\begin{aligned} & \text{theoremCorrectnessTimeCheck} : (\phi : \text{StackPredicate})(time_1 : \text{Time}) \\ & \rightarrow \langle \text{stackPred2SPred} (\text{timeCheckPreCond}\ time_1 \wedge sp\ \phi) \rangle^{iff} \text{checkTimeScript}^b\ time_1 \\ & \quad \langle \text{stackPred2SPred}\ \phi \rangle \end{aligned}$$

Now we can determine the weakest precondition for the combined script and prove its correctness as follows:

$$\begin{aligned} & \text{theoremCorrectnessCombinedMultiSigTimeCheck} : (time_1 : \text{Time}) (pbk1\ pbk2\ pbk3\ pbk4 : \mathbb{N}) \\ & \rightarrow \langle \text{stackPred2SPred} (\text{timeCheckPreCond}\ time_1 \wedge sp \\ & \quad \text{weakestPreCondMultiSig-2-4}^s\ pbk1\ pbk2\ pbk3\ pbk4) \rangle^{iff} \\ & \quad \text{checkTimeScript}^b\ time_1 ++ \text{multiSigScript2-4}^b\ pbk1\ pbk2\ pbk3\ pbk4 \\ & \quad \langle \text{acceptState} \rangle \end{aligned}$$

```

theoremCorrectnessCombinedMultiSigTimeCheck time1 pbk1 pbk2 pbk3 pbk4 =
  stackPred2SPred (timeCheckPreCond time1 ∧sp
    weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4)
    <><>< checkTimeScriptb time1 >>< theoremCorrectnessTimeCheck
      (weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4) time1 >
  stackPred2SPred (weakestPreCondMultiSig-2-4s pbk1 pbk2 pbk3 pbk4)
    <><>< multiSigScript2-4b pbk1 pbk2 pbk3 pbk4
      >>< theoremCorrectnessMultiSig-2-4 pbk1 pbk2 pbk3 pbk4 >e
  stackPred2SPred acceptStates ■p

```

The weakest precondition states that the state time is $\geq time_1$, and that the weakest precondition for the multisig script is fulfilled ($\wedge sp$ forms the conjunction of the two conditions). For proving it we used a combination of both methods, the second method was used to determine preconditions for the two parts of the scripts, and the first method, where we used whole scripts instead of basic instructions, was used to determine the combined weakest precondition.

7 Using Agda to Determine Readable Weakest Preconditions

Our library provides the operational semantics for (a subset of) Bitcoin SCRIPT, and a framework for specifying and reasoning about weakest preconditions. The Agda user has to specify the script to be verified, and then consider suitable pieces of the specified script and provide weakest preconditions. Agda will then create goals, which are unimplemented holes in the code. Agda will display the type of goals and list of assumptions available for solving them, and provide considerable additional support for resolving those goals. For instance, it allows to refine partial solutions provided by the user by applying it to sufficiently many new goals. Agda will as well automatically create case distinctions (such as whether an element of type `Maybe` is `just` or `nothing`). Agda can solve goals if the solution is unique and can be found in a direct way. Agda's automated theorem proving support for finding solutions which are not unique is not very strong due to the high complexity of the language.

Agda Reflection [3] is an ongoing project which already now provides a considerable library for inspecting code inside a goal and computing solutions as Agda code. The aim is to provide something similar to Coq's tactic language. In our code we frequently had to consider a nested case distinction for proving a goal, where most cases were solved because at one point one of the arguments became an element of the empty type. Automating this using Agda Reflection would make it much easier to use our library.

Finding a description of the weakest precondition has to be done manually at the moment. We plan to create a library which computes such descriptions for instructions or small pieces of instructions. Sometimes it is easier to provide weakest precondition for small pieces of code, for instance in case of the multisig instruction the weakest precondition for the instruction itself is very complex, whereas the weakest precondition for the P2MS script is much easier to display. Defining and simplifying the weakest preconditions in the intermediate steps has to be done manually at the moment. Proofs have to be done manually in Agda, but they are relatively easy because of Agda's support for developing proofs. It would be desirable to have a more automated support, where the user only needs to specify the verification conditions, but proofs are carried out automatically. In general our impression is that for writing programs and specifying verification conditions Agda is very suitable: one obtains code which is very readable and close to standard mathematical notations. Where Agda is lacking is in providing support for machine assisted proofs of the resulting conditions.

Regarding the question, which of the two approaches to use (working backwards step-by-step or using symbolic evaluation), we have only some heuristics at the moment. A good approach is that for pieces of code, where one has an intuition what the underlying program written in Agda could be, the symbolic evaluation is more suitable. For longer code, a good strategy is to cut the code into suitable pieces, for which one can find a symbolic program and weakest preconditions, and then work oneself backwards using the first approach starting from the acceptance condition. Note that symbolic execution can be done very fast: The user postulates variables for the arguments, applies the functions to be evaluated to those postulated arguments and then executes Agda’s normalisation mechanism. Then the user needs to manually inspect the result to see which sub expression trigger the case distinction. It would be nice project to develop a procedure which automates that process of symbolic execution – this could be applicable to verification of other kinds of programs as well.

8 Conclusion

In this paper, we have implemented and tested two methods for developing human-readable weakest preconditions and proving their correctness. These methods can help smart contract developers to fill the validation gap between user requirements and formal specification. We have argued that weakest preconditions in Hoare logic is the correct notion for specifying the security property of access control. We have applied our approaches to P2PKH, P2MS, and a combination of P2MS with a time lock. The whole approach has been formalised in Agda [47].

In future work, we will treat non-local instructions such as `OP_IF`, `OP_ELSE`, and `OP_ENDIF`, and will formalise key instructions to extend our approach to the whole of Bitcoin `SCRIPT`. The difficulty is nesting of conditionals, and that Bitcoin scripts are not structured, and therefore some additional work needs to be done to find the matching of if-then-else instructions. In our approach, we will use an expanded state space for dealing with those conditionals. Moreover, we plan to expand our library to support finding weakest preconditions for scripts having conditionals in a modular way. Furthermore, we plan the make the process of script verification more user-friendly by using a text parser that can record the instructions used for verification.

In addition, we aim to generalise the verification of P2MS to arbitrary m out of n multiscripts, where the challenge is finding a suitable generic human-readable weakest precondition.

Another route for future research is to develop our approach into a framework for developing smart contracts that are correct by construction. One way to build such smart contracts is to use Hoare Type Theory [27, 39].

References

- 1 Adacore. SPARK 2014, retrieved 9 november 2021. URL: <https://www.adacore.com/about-spark>.
- 2 Agda Team. Agda documentation, retrieved 21 april 2022. URL: <https://agda.readthedocs.io/en/latest/index.html>.
- 3 Agda Team. Agda Reflection, retrieved 21 april 2022. URL: <https://agda.readthedocs.io/en/latest/language/reflection.html>.
- 4 Mouhamad Almakhour, Layth Sliman, Abed Ellatif Samhat, and Abdelhamid Mellouk. Verification of smart contracts: A survey. *Pervasive and Mobile Computing*, 67:1–19, 2020. doi:10.1016/j.pmcj.2020.101227.

- 5 Sidney Amani, Myriam Bégel, Maksym Bortin, and Mark Staples. Towards Verifying Ethereum Smart Contract Bytecode in Isabelle/HOL. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 66–77, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167084.
- 6 Andreas M Antonopoulos. *Mastering Bitcoin: Programming the open blockchain (Second ed.)*. O'Reilly Media, Inc., 2017.
- 7 Nicola Atzei, Massimo Bartoletti, Stefano Lande, and Roberto Zunino. A formal model of bitcoin transactions. In *Financial Cryptography and Data Security*, pages 541–560, Berlin, Heidelberg, 2018. Springer Berlin Heidelberg. doi:10.1007/978-3-662-58387-6_29.
- 8 Massimo Bartoletti and Roberto Zunino. BitML: A Calculus for Bitcoin Smart Contracts. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18*, pages 83–100, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3243734.3243795.
- 9 Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-Cho-Coq, a Framework for Certifying Tezos Smart Contracts. In *Formal Methods. FM 2019 International Workshops*, pages 368–379, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-54994-7_28.
- 10 Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, and Santiago Zanella-Béguelin. Formal Verification of Smart Contracts: Short Paper. In *Proceedings of the 2016 ACM Workshop on Programming Languages and Analysis for Security, PLAS '16*, pages 91–96, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2993600.2993611.
- 11 Bitcoin Community. Welcome to the Bitcoin Wiki. Available from <https://en.bitcoin.it/wiki/Script>, 2010.
- 12 Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda – A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics*, pages 73–7, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-03359-9_6.
- 13 Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014. URL: <https://ethereum.org/en/whitepaper>.
- 14 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, and Philip Wadler. Native Custom Tokens in the Extended UTXO Model. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 89–111, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61467-6_7.
- 15 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Jann Müller, Michael Peyton Jones, Polina Vinogradova, Philip Wadler, and Joachim Zahnentferner. UTXOma: UTXO with Multi-asset Support. In *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, pages 112–130, Cham, 2020. Springer International Publishing. doi:10.1007/978-3-030-61467-6_8.
- 16 Manuel M. T. Chakravarty, James Chapman, Kenneth MacKenzie, Orestis Melkonian, Michael Peyton Jones, and Philip Wadler. The Extended UTXO Model. In *Financial Cryptography and Data Security*, pages 525–539, Cham, 2020. Springer International Publishing.
- 17 James Chapman, Roman Kireev, Chad Nester, and Philip Wadler. System F in Agda, for Fun and Profit. In *Mathematics of Program Construction*, pages 255–297, Cham, 2019. Springer International Publishing. doi:10.1007/978-3-030-33636-3_10.
- 18 Martin Churchill, Peter D. Mosses, Neil Sculthorpe, and Paolo Torrini. Reusable components of semantic specifications. In Shigeru Chiba, Éric Tanter, Erik Ernst, and Robert Hirschfeld, editors, *Transactions on Aspect-Oriented Software Development XII*, pages 132–179, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. doi:10.1007/978-3-662-46734-3_4.
- 19 Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Comput. Surv.*, 28(4):626–643, December 1996. doi:10.1145/242223.242257.

- 20 crypto.stackexchange. Is there any famous protocol that were proven secure but whose proof was wrong and lead to real world attacks?, retrieved 22 april 2022. URL: <https://crypto.stackexchange.com/questions/98829/is-there-any-famous-protocol-that-were-proven-secure-but-whose-proof-was-wrong-a>.
- 21 Peter D and Mosses. Modular structural operational semantics. *Journal of Logic and Algebraic Programming*, 60-61(0):195–228, 2004. doi:10.1016/j.jlap.2004.03.008.
- 22 Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975. doi:10.1145/360933.360975.
- 23 Etherscan. TheDAO smart contract 2016, retrieved 27 march 2022. Available from <http://etherscan.io/address/0xbb9bc244d798123fde783fcc1c72d3bb8c189413#code>.
- 24 Jean-Christophe Filliâtre and Andrei Paskevich. Why3 — Where Programs Meet Provers. In *Programming Languages and Systems*, pages 125–128, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. doi:10.1007/978-3-642-37036-6_8.
- 25 Yoichi Hirai. Defining the Ethereum Virtual Machine for Interactive Theorem Provers. In *Financial Cryptography and Data Security*, pages 520–535, Cham, 2017. Springer International Publishing. doi:10.1007/978-3-319-70278-0_33.
- 26 C. A. R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–585, October 1969. doi:10.1145/363235.363259.
- 27 IMDEA Software Institute. HTT: Hoare Type Theory, 10 march 2015. Available from <https://software.imdea.org/~aleks/htt/>.
- 28 Sukrit Kalra, Seep Goel, Mohan Dhawan, and Subodh Sharma. ZEUS: Analyzing Safety of Smart Contracts. In *25th Annual Network and Distributed System Security Symposium, NDSS 2018, San Diego, California, USA, February 18-21, 2018*, pages 1–15. The Internet Society, 2018. doi:10.14722/ndss.2018.23082.
- 29 James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976. doi:10.1145/360248.360252.
- 30 Rick Klomp and Andrea Bracciali. On Symbolic Verification of Bitcoin’s script Language. In *Data Privacy Management, Cryptocurrencies and Blockchain Technology*, pages 38–56, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-030-00305-0_3.
- 31 Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making Smart Contracts Smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, CCS ’16*, pages 254–269, New York, NY, USA, 2016. Association for Computing Machinery. doi:10.1145/2976749.2978309.
- 32 Luiz Eduardo G. Martins and Tony Gorschek. Requirements engineering for safety-critical systems: Overview and challenges. *IEEE Software*, 34(4):49–57, 2017. doi:10.1109/MS.2017.94.
- 33 Anastasia Mavridou, Aron Laszka, Emmanouela Stachtari, and Abhishek Dubey. VeriSolid: Correct-by-Design Smart Contracts for Ethereum. In *Financial Cryptography and Data Security*, pages 446–465, Cham, 2019. Springer International Publishing.
- 34 Orestis Melkonian. Formalizing BitML Calculus in Agda, 2019. Student Research Competition, Poster Session, ICFP’19. URL: <https://omelkonian.github.io/data/publications/formal-bitml.pdf>.
- 35 Orestis Melkonian. Formalizing Extended UTxO and BitML Calculus in Agda. Master’s thesis, Utrecht University, Department of Information and Computing Sciences, July 2019. URL: <https://studenttheses.uu.nl/handle/20.500.12932/32981>.
- 36 Dominic P. Mulligan, Scott Owens, Kathryn E. Gray, Tom Ridge, and Peter Sewell. Lem: Reusable engineering of real-world semantics. *ACM SIGPLAN Notices*, 49(9):175–188, August 2014. doi:10.1145/2692915.2628143.
- 37 Yvonne Murray and David A. Anisi. Survey of Formal Verification Methods for Smart Contracts on Blockchain. In *2019 10th IFIP International Conference on New Technologies, Mobility and Security (NTMS)*, pages 1–6, 2019. doi:10.1109/NTMS.2019.8763832.

- 38 Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized Business Review*, 2008. URL: <https://www.debr.io/article/21260.pdf>.
- 39 Nanevski, Aleksandar and Vafeiadis, Viktor and Berdine, Josh. Structuring the Verification of Heap-Manipulating Programs. *SIGPLAN Not.*, 45(1):261–274, January 2010. doi:10.1145/1707801.1706331.
- 40 Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming: 6th International School, AFP 2008, Heijten, The Netherlands, May 2008, Revised Lectures*, pages 230–266, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. doi:10.1007/978-3-642-04652-0_5.
- 41 Russell O’Connor. Simplicity: A new language for blockchains. In *Proceedings of the 2017 Workshop on Programming Languages and Analysis for Security, PLAS ’17*, pages 107–120, New York, NY, USA, 2017. Association for Computing Machinery. doi:10.1145/3139337.3139340.
- 42 Daejun Park, Yi Zhang, Manasvi Saxena, Philip Daian, and Grigore Roşu. A Formal Verification Tool for Ethereum VM Bytecode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2018*, pages 912–915, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3236024.3264591.
- 43 Christine Paulin-Mohring. *Introduction to the Coq Proof-Assistant for Practical Software Verification*, pages 45–95. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012. doi:10.1007/978-3-642-35746-6_3.
- 44 Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. *The Evolution of Forth*, pages 625–670. Association for Computing Machinery, New York, NY, USA, 1996. doi:10.1145/234286.1057832.
- 45 Maria Ribeiro, Pedro Adão, and Paulo Mateus. *Formal Verification of Ethereum Smart Contracts Using Isabelle/HOL*, pages 71–97. Springer International Publishing, Cham, 2020. doi:10.1007/978-3-030-62077-6_7.
- 46 Anton Setzer. Modelling Bitcoin in Agda. *CoRR*, abs/1804.06398, 2018. arXiv:1804.06398.
- 47 Anton Setzer, Fahad Alhabardi, and Bogdan Lazar. Verification Of Smart Contracts With Agda. Available from <https://github.com/fahad1985lab/Smart--Contracts--Verification--With--Agda>, 2021.
- 48 Stack Exchange Inc. provable security - Is there any famous protocol that were proven secure but whose proof was wrong and lead to real world attacks? , retrieved 22 april 2022. Availabe from <https://crypto.stackexchange.com/questions/98829/is-there-any-famous-protocol-that-were-proven-secure-but-whose-proof-was-wrong-a>.
- 49 Philip Wadler, Wen Kokke, and Jeremy G. Siek. *Programming Language Foundations in Agda*. Online textbook, July 2020. URL: <https://plfa.github.io/Equality/>.
- 50 Maximilian Wohrer and Uwe Zdun. Smart Contracts: Security Patterns in the Ethereum Ecosystem and Solidity. In *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pages 2–8, 2018. doi:10.1109/IWBOSE.2018.8327565.

Formalisation of Dependent Type Theory: The Example of CaTT

Thibaut Benjamin   

Université Paris-Saclay, CEA, List, F-91120, Palaiseau, France

Abstract

We present the type theory CaTT, originally introduced by Finster and Mimram to describe globular weak ω -categories, formalise this theory in the language of homotopy type theory and discuss connections with the open problem internalising higher structures. Most of the studies about this type theory assume that it is well-formed and satisfy the usual syntactic properties that dependent type theories enjoy, without being completely clear and thorough about what these properties are exactly. We use our formalisation to list and formally prove all of these meta-properties, thus filling a gap in the foundational aspect. We discuss the aspects of the formalisation inherent to CaTT. We present the formalisation in a way that not only handles the type theory CaTT but also related type theories that share the same structure, and in particular we show that this formalisation provides a proper ground to the study of the theory MCaTT which describes the globular monoidal weak ω -categories. The article is accompanied by a development in the proof assistant Agda to check the formalisation that we present.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory; Theory of computation \rightarrow Categorical semantics

Keywords and phrases Dependent type theory, homotopy type theory, higher categories, formalisation, Agda, proof assistant

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.2

Related Version *arXiv Version*: <https://arxiv.org/abs/2111.14736>

Supplementary Material *Software (Agda Source Code)*: <https://github.com/thibautbenjamin/catt-formalization>; archived at [swh:1:dir:db3f3e900b7541e07a7f19b1d895daa7fedbadd5](https://swh.io/dir/db3f3e900b7541e07a7f19b1d895daa7fedbadd5)

Acknowledgements I want to thank Samuel Mimram and Eric Finster for their guidance in this project and the anonymous reviewers for their helpful comments.

1 Introduction

This article presents and formalises the foundations of the type theory CaTT introduced by Finster and Mimram [13], and similar type theories. CaTT is designed to encode a flavour of higher categorical structures called weak ω -categories, and its semantics has been proved [9] equivalent to a definition of weak ω -categories due to Maltsiniotis [22] based on an approach by Grothendieck [14]. Another example that fits in our framework is the theory MCaTT [8], which models monoidal weak ω -categories. We have formalised the work we present the proof-assistant Agda¹ in a fully proof-relevant way, without using Axiom K. we rely strongly on insights and ideas that emerged with Homotopy Type Theory (HoTT), and follow the homotopical interpretation of identity types. So we call this setting HoTT as well.

Although a few of the aforementioned articles primarily focus on CaTT, none of them give a complete foundation for it. Instead they simply assume that some syntactic meta-properties are satisfied. Far from being a shortcoming of those articles, this is common practice in the

¹ <https://github.com/thibautbenjamin/catt-formalization/tree/TYPES2021>



© Thibaut Benjamin;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 2; pp. 2:1–2:21

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

type theory community. Indeed low-level descriptions are lengthy and it is usually accepted that a full description satisfying the usual meta-theoretic properties is possible. As a result these articles rely on the reader's ability to infer and accept these foundations.

Moreover, the goal of formalising dependent type theory within itself, has been a long standing goal of dependent type theory [11], and is a hard and interesting problems. Important developments, such as the MetaCoq project [25], a formalisation of Coq written in Coq are being explored in this direction. More specifically in HoTT, it is a famous open problem to give an definition of HoTT within itself [24], and it is known to be closely linked with another open problem: The study of higher algebraic structures in HoTT.

We first present two main approaches to formalise dependent type theories and explore some connections with the definition of higher algebraic structures. We elaborate on the approach we chose and how it avoids the difficulty by restricting our focus to a purely syntactic definition. We then give a quick informal presentation of the theory CaTT and of a simpler type theory called GSeTT. Next, we discuss the formalisation of GSeTT, along with its meta-theoretic properties. Finally, we introduce our formalism of globular type theory, that models type theories similar to GSeTT, and show the meta-theoretic properties of interest. We show that CaTT and MCaTT are both instances of globular type theories and thus we get those properties for free. We present our formalisation in Agda pseudo-code with the convention that all the free variables are implicitly universally quantified, and provide some explanation for the syntax to help the reader navigate the article and the formalisation.

2 Structural foundation of dependent type theory

In this section, we give informal presentation of the type theories that we consider, and discuss the ways they can be formalised. The type theories we study are centred around four kinds of object, that we introduce here along with corresponding notations

$$\begin{array}{ll} \text{contexts: } & \Gamma, \Delta, \dots \\ \text{types: } & A, B, \dots \end{array} \qquad \begin{array}{ll} \text{terms: } & t, u, \dots \\ \text{substitutions: } & \gamma, \delta, \dots \end{array}$$

Each of these object is associated to a well-formedness judgement

$$\begin{array}{ll} \Gamma \text{ is a valid context: } & \Gamma \vdash \\ t \text{ is a term of type } A \text{ in } \Gamma: & \Gamma \vdash t : A \\ A \text{ is a valid type in } \Gamma: & \Gamma \vdash A \quad \gamma \text{ is a valid substitution from } \Delta \text{ to } \Gamma: \quad \Delta \vdash \gamma : \Gamma \end{array}$$

There are lots of flavours of type theories: dependent, linear (in the sense of linear logic), graded, or with refinement types... We consider are dependent type theories similar to Martin-Löf type theory without computation rules, that all respect the following rules.

$$\begin{array}{c} \frac{}{\emptyset \vdash} \text{(EC)} \qquad \frac{\Gamma \vdash A}{\Gamma, x : A \vdash} \text{(CE) \quad Where } x \notin \text{Var}(\Gamma) \\ \frac{\Gamma \vdash (x : A) \in \Gamma}{\Gamma \vdash x : A} \text{(VAR)} \\ \frac{\Delta \vdash}{\Delta \vdash \langle \rangle : \emptyset} \text{(ES)} \qquad \frac{\Delta \vdash \gamma : \Gamma \quad \Gamma, x : A \vdash \quad \Delta \vdash t : A[\gamma]}{\Delta \vdash \langle \gamma, x \mapsto t \rangle : (\Gamma, x : A)} \text{(SE)} \end{array}$$

Where $t[\gamma]$ is the application of the substitution γ to the term t and $\text{Var}(\Gamma)$ is the set of variables in Γ . For the sake of conciseness, we implicitly assume all type theories to satisfy those rules.

2.1 The typed syntax approach

An approach proposed by Dybjer [12] for internalising the semantics of dependent type theory is to define a typed syntax by induction. In this approach, every syntactic object must be well-formed by construction, it makes no sense to even consider an ill-formed object. For this reason we also refer to this approach as the *intrinsic type theory*.

Structure of the typed syntax. The intrinsic formalisation of dependent type theory relies on four inductive inductive types. They correspond to the four kinds of syntactic objects. We only give their signature in Agda pseudo-code to discuss specific points here. We use the keyword `data` to introduce mutually inductive types and the type `Set` is the type of types without any truncation assumption, this conflicts with the HoTT terminology.

```
data Ctx : Set
data Ty : Ctx → Set
data Tm : ∀ Γ → Ty Γ → Set
data Sub : Ctx → Ctx → Set
```

In this framework the type `Ctx` is the type of well-formed contexts i.e., the type of all derivation trees of judgements of the form $\Gamma \vdash$, and similarly for the three other types. The type of types `Ty` depends on the type `Ctx`: It does not make sense to require a type to be well-formed, without a reference to the context in which this well-formedness is satisfied.

The semantics of the theory. This approach was originally proposed by Dybjer [12], in order to internalise a categorical device called category with families (CwF). It captures the algebraic structure of dependent type theory, and allows for studying its semantics. Defining the theory along its semantics is an important goal as it allows to study complex meta-theoretic properties, such as the independence of certain axioms by forcing or the initiality conjecture. This conjecture states that a theory should realise a CwF that is initial among a class of CwF that support the constructions of the theory. The typed syntax approach, provided that it can be developed completely, would provide a definition of a type theory that by construction enjoys the structure of a CwF and comes with a suitable notion of models by using the induction principle. Thus, it provides a unification between syntax and semantics, and gives both completely.

The dependency issue. An intrinsic approach presents hard challenges. For instance, we expect the action of substitutions on types and terms do be functorial:

$$A[\gamma][\delta] = A[\gamma \circ \delta] \quad t[\gamma][\delta] = t[\gamma \circ \delta]$$

Within the typed syntax, the second of these equations is not even a valid statement. Indeed, suppose given the term $t : \mathbf{Tm} \ \Gamma \ \mathbf{A}$ and two substitutions $\gamma : \mathbf{Sub} \ \Delta \ \Gamma$ and $\delta : \mathbf{Sub} \ \Theta \ \Delta$ then the term $t[\gamma \circ \delta]$ is of type $\mathbf{Tm} \ \Theta \ \mathbf{A}[\gamma \circ \delta]$, whereas the term $t[\gamma][\delta]$ is of type $\mathbf{Tm} \ \Theta \ \mathbf{A}[\gamma][\delta]$. Although these two types are equal by application of the first equality, they are not definitionally so. Thus even stating this equality, requires transporting one of the terms along the equality between the types. Proving later properties then requires a proper handling of these transports which becomes quickly prohibitive.

The coherence problem. On top the aforementioned technical issue, there is a more fundamental obstruction called the *coherence problem*: All the equalities, like the ones mentioned, have to be assumed as part of the structure in the form of term witnesses. But this is not enough, as equalities between those witnesses also need to be specified, and then

equalities between their witnesses, and so on. This leads to a seemingly infinite amount of equations needed to present the theory. In such cases, it is sometimes possible to generate all of them with a finite scheme, but it is an open question to determine whether it is possible in this case. The same coherence issue also appears in a widely studied open problem consisting in defining higher algebraic structures in HoTT. In HoTT, each type is naturally endowed with a higher structure of groupoid, thanks to the identity types, and internalising a higher algebraic structure requires an additional higher structure. The coherence problem amounts to listing all the axioms of the latter in terms of the former, and ensuring that they play nicely together. A particularly studied instance of this is the definition of semi-simplicial types [16], which has been open since the birth of HoTT.

Workarounds for those problems. Several techniques to avoid the coherence problem while formalising either higher algebraic structures or dependent type theories have been considered. However, none of them solve the problem which is still open.

- In his work (anterior to HoTT) to formalise dependent type theory within itself [11], Chapman sidesteps the dependency by considering a heterogeneous equality predicate and mostly ignores the coherence problem. More recently, Lafont adapted this idea ² and could avoid the coherence problem by requiring uniqueness of identity proofs (UIP) partially, which is not compatible with HoTT. This development is beyond the capabilities of Agda, and requires using the option `--no-termination-check` which deactivates the termination checker and is unsound in Agda.
- A technique proposed by Altenkirch, Capriotti and Kraus [2] is to use a 2-level type theory, where the equality is strict, but there exists a sub class of types called fibrant which also have a weak equality. Then one can restrict the focus only on fibrant types in the semantics to recover the desired behaviour. This lead to a lot of promising developments both for studying higher structures and formalising HoTT [18], but it sidesteps the coherence problem with a strict equality type, and thus is not internal to HoTT. This approach was also considered by Lafont in the same project³.
- Another approach developed by Altenkirch and Kaposi [3] for formalising a dependent type theory consists in using quotient inductive inductive types (QIITs), a type theoretic construction that allows for set-truncated higher inductive types. The semantical interpretation of QIITs is very intricate, and from a practical perspective, it is really hard to develop such a method: Higher inductive types are not natively supported in Agda, and using them requires defining recursors manually and carefully avoiding Agda's pattern matching mechanism for those types, for which it is unsound.

Overall, the typed syntax approach presents an important challenge in the form of the coherence problem, so it does not translate easily in HoTT. The works we have presented provide workarounds by constraining the meta-theory further and adding some amount of proof-irrelevance, but this prevents the internalisation of the semantics.

2.2 The raw syntax approach

In this article, we take an alternate route to formalising dependent type theory, based on the separation of the syntax and the rules of the theory. To highlight the difference we call *raw syntax* the syntactic elements that are not tied to a derivation tree. Contrary to the typed

² <https://github.com/amblafont/omegatt-agda/tree/master>

³ <https://github.com/amblafont/omegatt-agda/tree/2tt-fibrant>

syntax, raw syntax may contain ill-formed entities that do not correspond to any entity of the theory. We delegate the computational duty to the raw syntax, which completely sidesteps the coherence problem. We call this approach an *extrinsic* formalisation of a dependent type theory since it requires combining two separate ingredients, the raw syntax and the judgements. In this article we present a formalisation of the dependent type theory `CaTT` in an extrinsic way. Similar formalisations are being developed, notably by Finster⁴ and Rice⁵, but with the intent of formalising variations of `CaTT` with more intricate type-theoretic constructions. Moreover Lafont, Hirshowitz and Tabareau have also developed a formalisation of a type theory related to `CaTT` in a similar fashion in `Coq`⁶ [19] with the goal of internalising that types are weak ω -groupoids [21, 27, 4].

Variables management. In order to compute operations on the raw syntax, we need the variables to be identifiable. To avoid the need of quotients, we develop a foundation in which the variables are natively normalised: Each variable name is uniquely determined. We use a variation on De Bruijn levels: The type of variables is the type of natural numbers \mathbb{N} and we require contexts to enumerate their variables in increasing order. Since there is no variable binder in the theory `CaTT`, this suffices to determine all the variables.

Structure of the raw syntax. We now present the foundational structure of the dependent type theories that we are interested in. We first show the empty type theory: A theory with all the required structure but no types. This theory is completely vacuous: Its only context is empty, and no term or type is derivable in it. It is not part of our actual project and is not of any interest in itself, but we present it here to emphasize the structure of our type theories and factor out the common features of the theories we study. We first define the raw syntax: Contexts and substitutions may be built out of variables types and terms, and any variable is also a term. Type constructors and other term constructors may vary depending on the theory, so we do not include them in the empty type theory.

► **Definition 1.** *We define the raw syntax of the empty dependent type theory as a collection of four non-dependent types defined by mutual induction, representing respectively the (raw) contexts, substitutions, terms and types*

```

data Pre-Ctx : Set where
  () : Pre-Ctx
  _#_ : Pre-Ctx → ℕ → Pre-Ty → Pre-Ctx
data Pre-Sub : Set where
  <> : Pre-Sub
  <_ _> : Pre-Sub → ℕ → Pre-Tm → Pre-Sub
data Pre-Ty where
data Pre-Tm where
  Var : ℕ → Pre-Tm

```

The constructor `Var` produces an inhabitant of the type `Pre-Tm` from a variable (of type \mathbb{N}), and there is no constructor for the type `Pre-Ty` since there is no type. Those types do not need to be mutually inductive here, but we define them as such by anticipation with later theories. For all intents and purposes, we can think of contexts (resp. substitutions) as lists of pairs of the form (x, A) where x is a variable and A is a type (resp. lists of pairs (x, t) where x is a variable and t is a term, any variable is also a term). We denote $\ell \Gamma$ the length of a context Γ .

⁴ <https://github.com/ericfinster/catt.io/tree/master/agda>

⁵ <https://github.com/alexarice/catt-agda>

⁶ <https://github.com/amblafont/weak-cat-type/tree/untyped2tt>

The action of substitutions. We define these operations on the raw syntax levels, and they are the reason why we need to introduce variable names. Those are functions that compute a new syntactic entity from a given one and a substitution to apply to it. These operations compute to normal forms, so when two applications are equal, the results of the computations are definitionally so, thus we do not need equality witnesses and avoid the coherence problem at this level. The composition is an action of substitutions on substitutions.

► **Definition 2.** We define the action of substitutions on types, terms and substitutions as the following mutually inductive operations

```

_[]T : Pre-Ty → Pre-Sub → Pre-Ty
_[]t : Pre-Tm → Pre-Sub → Pre-Tm
_[]_ : Pre-Sub → Pre-Sub → Pre-Sub

```

```
() [ γ ]T
```

```
Var x [ <> ]t = Var x
```

```
Var x [ < γ , v ↦ t > ]t = if x ≡ v then t else ((Var x) [ γ ]t)
```

```
<> ∘ δ = <>
```

```
< γ , x ↦ t > ∘ δ = < γ ∘ δ , x ↦ t [ δ ]t >
```

Where $() [\gamma]T$ represents the empty case, since there are no constructors for the type `Pre-Ty`. Again we use mutually inductive types for consistency with later developments.

Judgements and inference rules. We now select the well-formed syntactic entities from the syntax by introducing the judgements together with their inference rules. We define those as mutually inductive predicates over the raw syntax. We prove later (Theorems 17 and 25) that those types are actually propositions, their signature is given by

```

data _⊢C : Pre-Ctx → Set
data _⊢T : Pre-Ctx → Pre-Ty → Set
data _⊢t#_ : Pre-Ctx → Pre-Tm → Pre-Ty → Set
data _⊢S>_ : Pre-Ctx → Pre-Sub → Pre-Ctx → Set

```

The type $\Gamma \vdash_C$, for instance, is meant to represent the judgement $\Gamma \vdash$, an inhabitant of this type is built out of the type constructors, corresponding to inference rules. Hence an element of this type can be thought of as a derivation tree. Reasoning by induction on derivation trees, a common technique to prove meta-theoretic properties of dependent type theory, just translates to reasoning by induction on those four types. This discussion holds because there is no computation rules (i.e., rules postulating definitional equalities) in our theories: In the presence of such rules, one would need to consider higher inductive types, in order to preserve the correspondence between terms and derivation trees. The computation rules would then be higher order constructors.

► **Definition 3.** We define the following inference rules for constructing contexts, substitutions and variable terms, which are the inference rules of the empty type theory (again, the type `_⊢T` has no constructor, the theory is vacuous).

```

data _⊢C where
  ec : ∅ ⊢C
  cc : Γ ⊢C → Γ ⊢T A → x == ℓ Γ → (Γ · x # A) ⊢C

```



```

data _⊢T_ where
data _⊢t_#_ where
  var : Γ ⊢C → x # A ∈ Γ → Γ ⊢t (Var x) # A
data _⊢S_>_ where
  es : Δ ⊢C → Δ ⊢S <> > ∅
  sc : Δ ⊢S γ > Γ → (Γ · x # A) ⊢C → (Δ ⊢t t # (A [ γ ]T))
      → x == y → Δ ⊢S < γ , y ↦ t > > Γ · x # A

```

A term constructor of one of these types corresponds to a rule in the theory, so we have given the same name to the term constructors and the corresponding rules. In the rules `cc` and `sc`, we consider an equality on variables as part of the required data for a rule. This might seem odd at first, as we could instead inline this equality. However this lets us eliminate on the equality only when needed, which is important to avoid the axiom K. In the rule `cc`, the condition `x == ℓ Γ` enforces that contexts enumerate their variables in order.

Raw vs. typed syntax. Using a raw syntax approach has many advantages over a typed syntax. First, it can be completely formalised in a proof-relevant way and does not require any truncation. It also allows the use of the formalisation as a certified type checker, which is important for practical purposes. However, it focuses more narrowly on the syntax of the theory, and does not allow to internalise its algebraic properties (the CwF structure) nor its semantics. The transition from typed syntax is reminiscent of a more general construction to handle induction induction described by Kaposi, Kovács and Lafont [17].

Absence of semantics. This article focuses strongly on the syntax of the theory, and lacks semantics: We do not provide a way to show that a type models a theory. We conjecture that there should exist such a way, in the form of an internal notion of CwF, but that defining this notion would amount to solving the coherence problem. We do not address this question in this article. However we show that the syntax we provide realises an instance of this tentative notion of internal CwF, in which a lot of required equalities hold on the nose. Of course since this notion does not exist, this statement is imprecise, but we verify numerous properties that are expected to be part of this structure, such as weakening admissibility, the functoriality of the action of substitutions on types and terms... We also show the uniqueness of derivations, which is an important part of this tentative structure, as well as the decidability of type checking, which is relevant for practical applications and a result about a classifications of the types and terms of our theories. A similar approach was developed by Abel, Öman and Vezzosi [1], where they considered a more complex theory to prove weaker results, and give a similar discussion about the typed syntax in conclusion. We consider a simpler theory (in particular without computation rule), and are able to prove more results about it.

The duality theory/higher structure. We have discussed the coherence problem, and that it is the meeting point between internalising type theory and higher structures. Our work on `CaTT` illustrates the connection: The formalisation of `CaTT` can be seen as a formalisation of the internal language of weak ω -categories, since those two are the same [9]. More generally, we conjecture that the coherence problem appears in higher structures, like in type theory, at the level of the models and that in many case it is possible to define the internal language of a structure without running into it.

3 Introduction to the theory CaTT

We present here the type theory CaTT, focusing mostly on the syntactic aspects. We still provide some semantical intuition and we refer the reader to existing articles [13, 9] for more in-depth discussions about the semantics. In this section, we provide an informal presentation to serve as guideline for our foundations.

3.1 The theory GSeTT

We first introduce the theory GSeTT which is simpler than the theory CaTT and serves as a basis on which this theory relies. In the theory GSeTT there are no term constructors, hence the only terms are the variables. There are two type constructors, that we denote \star and \rightarrow and which are subject to the following introduction rules

$$\frac{\Gamma \vdash}{\Gamma \vdash \star} (\star\text{-INTRO}) \qquad \frac{\Gamma \vdash A \quad \Gamma \vdash t : A \quad \Gamma \vdash u : A}{\Gamma \vdash t \xrightarrow[A]{} u} (\rightarrow\text{-INTRO})$$

The contexts of the theory GSeTT can be represented by *globular sets*. Those are analogues to graphs, except that they are allowed to have arrows in every dimension (often called *cells*). A cell of dimension $n + 1$ has for source and target a pair of cells of dimension n which are required to share the same source and target. In the type theory, this is imposed by rule (\rightarrow -INTRO) in which the terms t and u have to share the type A . We denote $\dim A$ the dimension of the cell represented by A . It can be computed as the number of iterated \rightarrow needed to write it. We denote $\dim \Gamma$ the maximal dimension of the types that appear in Γ .

► **Example 4.** We illustrate the correspondence between contexts and finite globular sets with a few examples, using a diagrammatic representation of globular sets where we give the same name to variables and their corresponding cells.

$$\begin{array}{l} \Gamma_c = (x : \star, y : \star, f : x \xrightarrow{\star} y, z : \star, h : y \xrightarrow{\star} z) \\ \Gamma_w = (x : \star, y : \star, f : x \xrightarrow{\star} y, g : x \xrightarrow{\star} y, \alpha : f \xrightarrow{x \rightarrow y} g, z : \star, h : y \xrightarrow{\star} z) \\ \Gamma_\circ = (x : \star, f : x \xrightarrow{\star} x) \end{array} \qquad \begin{array}{l} \begin{array}{c} x \xrightarrow{f} y \xrightarrow{h} z \\ \bullet \qquad \bullet \qquad \bullet \end{array} \\ \begin{array}{c} x \xrightarrow{f} y \xrightarrow{h} z \\ \downarrow \alpha \\ x \xrightarrow{g} y \xrightarrow{h} z \\ \bullet \qquad \bullet \qquad \bullet \end{array} \\ \begin{array}{c} x \\ \bullet \xrightarrow{f} \bullet \end{array} \end{array}$$

This correspondence is not an actual bijection: Several context may correspond to the same globular set if they only differ by reordering of the variables. One can account for this by considering the category of contexts with substitutions (called syntactic category). It is equivalent to the opposite of the category of finite globular sets [9, Th. 16]. A judgement $\Gamma \vdash x : A$ in the theory GSeTT corresponds to a well-defined cell x in the globular set corresponding to Γ , and the type A provides all the iterated sources and targets of x .

3.2 Ps-contexts

In the type theory CaTT there is an additional judgement on contexts, that recognises a special class of contexts called *ps-contexts*. We denote this judgement $\Gamma \vdash_{\text{ps}}$, and we introduce it with the help of an auxiliary judgement $\Gamma \vdash_{\text{ps}} x : A$. These two judgements are subject to the following derivation rules

$$\begin{array}{c}
\frac{}{x : \star \vdash_{\text{ps}} x : \star} \text{(PSS)} \\
\frac{\Gamma \vdash_{\text{ps}} f : x \xrightarrow{A} y}{\Gamma \vdash_{\text{ps}} y : A} \text{(PSD)} \\
\frac{\Gamma \vdash_{\text{ps}} x : A}{\Gamma, y : A, f : x \xrightarrow{A} y \vdash_{\text{ps}} f : x \xrightarrow{A} y} \text{(PSE)} \\
\frac{\Gamma \vdash_{\text{ps}} x : \star}{\Gamma \vdash_{\text{ps}}} \text{(PS)}
\end{array}$$

The semantical intuition is that \vdash_{ps} characterises the contexts corresponding to *pastings schemes* [5, 22]. Those are the finite globular sets defining an essentially unique composition in weak ω -categories. Finster and Mimram have given an alternate characterisation of the ps-contexts [13] using a relation on the variables of a context, denoted $x \triangleleft y$.

► **Definition 5.** *The relation \triangleleft in a context Γ , as the transitive closure of generated by*

$$x \triangleleft y \triangleleft z \text{ as soon as } \Gamma \vdash y : x \rightarrow z \text{ is derivable}$$

The authors have proved that a context is isomorphic to a ps-contexts if and only if this relation is a linear order on its variables.

Additionally, a ps-context Γ defines two subsets of its variables $\partial^-(\Gamma)$ and $\partial^+(\Gamma)$ respectively called the source and the target set, defined from a slightly more generic concept of i -source and i -target.

► **Definition 6.** *The i -source ∂_i^- and the i -target ∂_i^+ of a ps-context are given by induction*

$$\begin{array}{l}
\partial_i^-(x : \star) = (x : \star) \quad \partial_i^-(\Gamma, y : A, f : x \rightarrow y) = \begin{cases} \partial_i^-(\Gamma) & \text{if } \dim A \geq i \\ \partial_i^-(\Gamma), y : A, f : x \rightarrow y & \text{otherwise} \end{cases} \\
\partial_i^+(x : \star) = (x : \star) \quad \partial_i^+(\Gamma, y : A, f : x \rightarrow y) = \begin{cases} \partial_i^+(\Gamma) & \text{if } \dim A > i \\ \text{drop}(\partial_i^+(\Gamma)), y : A & \text{if } \dim A = i \\ \partial_i^+(\Gamma), y : A, f : x \rightarrow y & \text{otherwise} \end{cases}
\end{array}$$

where *drop* is the list with its head removed. The source (resp. target) of a ps-context Γ is the set of all variables in its $(\dim \Gamma)$ -source (resp. in its $(\dim \Gamma)$ -target), i.e., $\partial^\pm(\Gamma) = \partial_{(\dim \Gamma)}^\pm(\Gamma)$.

Semantically, those contain the variables in the source and target of the result of applying the essentially unique composition given by the pasting scheme corresponding to Γ .

► **Example 7.** The contexts Γ_c and Γ_w defined in Example 4 are ps-contexts, while the context Γ_\circ is not. We provide below the relation \triangleleft , and the source and target set variables.

$$\begin{array}{l}
\Gamma_c : \quad x \triangleleft f \triangleleft y \triangleleft h \triangleleft z \quad \partial^-(\Gamma_c) = \{x\} \quad \partial^+(\Gamma_c) = \{z\} \\
\Gamma_w : \quad x \triangleleft f \triangleleft \alpha \triangleleft g \triangleleft y \triangleleft h \triangleleft z \quad \partial^-(\Gamma_w) = \{x, f, y, h, z\} \quad \partial^+(\Gamma_w) = \{x, g, y, h, z\} \\
\Gamma_\circ : \quad x \triangleleft f \quad \partial^-(\Gamma_\circ) \text{ and } \partial^+(\Gamma_\circ) \text{ undefined}
\end{array}$$

3.3 The type theory CaTT

The type theory CaTT is obtained from the type theory GSeTT by adding new term constructors that witness the structure of weak omega-categories. There are two of these constructors, **op** and **coh**, so a term is either a variable or of the form **op** $_{\Gamma, A}[\gamma]$ or **coh** $_{\Gamma, A}[\gamma]$, where in both two last cases, Γ is a ps-context, A is a type and γ is a substitution. These term constructors are subject to the following introduction rules.

$$\frac{\Gamma \vdash_{\text{ps}} \quad \Gamma \vdash t : A \quad \Gamma \vdash u : A \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \text{op}_{\Gamma, t \rightarrow u}[\gamma] : t[\gamma] \rightarrow_{A[\gamma]} u[\gamma]} (\text{OP}) \quad \frac{\Gamma \vdash_{\text{ps}} \quad \Gamma \vdash A \quad \Delta \vdash \gamma : \Gamma}{\Delta \vdash \text{coh}_{\Gamma, A}[\gamma] : A[\gamma]} (\text{COH})$$

Both these rules apply only under extra side-conditions. We denote (C_{op}) the side condition of (OP) and (C_{coh}) the one of (COH). Those side conditions are the following (where $\text{Var}(A)$ denotes the set of variables needed to write the type A).

$$(C_{\text{op}}) : \begin{cases} \text{Var}(t) \cup \text{Var}(A) = \partial^-(\Gamma) \\ \text{Var}(u) \cup \text{Var}(A) = \partial^+(\Gamma) \end{cases} \quad (C_{\text{coh}}) : \text{Var}(A) = \text{Var}(\Gamma)$$

Recall that a ps-context is meant to represent an essentially unique composition in a weak ω -category. The rules (OP) and (COH) enforce this condition, in a weak sense analogue to requiring the type of composition to be contractible in HoTT. More specifically, the rule (OP) asserts that in a context Δ , for every situation described by a ps-context Γ (as witnessed by γ), there exists a term witnessing the existence of the composition of this situation. The rule (COH) imposes that any two such compositions are related by a higher cell. Indeed in this rule the type A is necessarily of the form $a \rightarrow b$, where a and b represent two ways of composing the ps-context. The role of the side condition is to prevent the composition to apply partially.

► **Example 8.** Consider the context Γ_c introduced in Example 4, we have established in Example 7 that it is a ps-context, with $\partial^-(\Gamma_c) = \{x\}$ and $\partial^+(\Gamma_c) = \{z\}$. The type $x \rightarrow z$ thus satisfies the condition (C_{op}) . This shows that for every context Δ with a substitution $\Delta \vdash \gamma : \Gamma_c$, we can define the term $\text{comp } \gamma = \text{op}_{\Gamma_c, x \rightarrow z}[\gamma]$, and we have a derivation of $\Delta \vdash \text{comp } \gamma : x[\gamma] \rightarrow z[\gamma]$. The data of a substitution $\Delta \vdash \gamma : \Gamma_c$ is equivalent to two composable arrows in Δ , and the semantics is that the term $\text{comp } \gamma$ witnesses the composition of those arrows. Additional examples are presented in [13].

4 Formalisation and properties of the theory GSeTT

From now on, we use the extrinsic formalisation of dependent type theory, and present our type theories in a similar way to the empty type theory. We first define GSeTT by specifying two constructors for the type Pre-Ty, corresponding to the two type constructors of GSeTT. We then prove the inversion and weakening lemmas, and show that this theory has a structure of a category with families, with particular contexts classifying its types and terms, that type checking is decidable, and the derivation trees are necessarily unique.

4.1 Formal presentation of the theory GSeTT

Recall that the type theory GSeTT has only two type constructors \star and \rightarrow , so we introduce two constructors to the type Pre-Ty. Note that this addition changes the entire raw syntax, since the other types Pre-Ctx, Pre-Tm and Pre-Sub are all mutually inductively defined with Pre-Ty. For simplicity purposes, we only present here the parts whose definition changes, but these changes actually propagate to the entire raw syntax.

► **Definition 9.** *The raw syntax of the type theory GSeTT is defined the same way as the raw syntax of the empty type theory, replacing the type Pre-Ty with the type*

```
-- Pre-Ctx, Pre-Sub and Pre-Tm defined like in the empty TT
data Pre-Ty where
  * : Pre-Ty
  _⇒[_]_ : Pre-Tm → Pre-Ty → Pre-Tm → Pre-Ty
```

Additionally, we introduce the action of substitution to be defined the same way on raw terms and substitutions as in the empty type theory, and we define it on raw types by

```
-- _[_]t and _o_ defined like in the empty TT
* [γ]T = *
(t ⇒[A] u)[γ]T = (t [γ]t) ⇒[A [γ]T] (u [γ]t)
```

We also specify the judgements of the theory, again those are defined in the same way for the empty type theory, except for the type judgement $_ \vdash_T _$. Since these judgements are mutually inductive, a change here again propagates to all the judgements of the theory.

► **Definition 10.** *The type theory GSeTT is obtained from its raw syntax by adding the judgements defined in the same way as for the empty type theory, except for the following*

```
-- _\vdash_C _\vdash_t\#_ and _\vdash_S>_ defined like in the empty TT
data _\vdash_T_ where
  ob : ∀ Γ → Γ \vdash_C → Γ \vdash_T *
  ar : ∀ Γ A t u → Γ \vdash_t t # A → Γ \vdash_t u # A → Γ \vdash_T t ⇒[A] u
```

We now present a lot of meta-properties that are required to study the semantics of the theory GSeTT. All of these properties are proved by induction on the derivation trees. A lot of these proofs are straightforward and we just give a discussion on the non-trivial proof techniques that come up. We refer the reader to the Agda implementation provided as supplementary material for the complete proofs of all of these properties.

4.2 Structure of the dependent type theory

First we show a few properties about the theory GSeTT, ensuring that the raw syntax together with a typing rule describe a dependent type theory with all the expected structure. Although these results are used a lot to study the semantics of the theory [13, 9], they are generally admitted and proving them requires as much specificity on the foundational aspects as we have provided here.

Weakening and judgement preservation. The following proposition states that GSeTT supports the weakening for types, terms and substitutions, as well as inversion.

► **Proposition 11.** *The theory GSeTT supports weakening for types, terms and substitutions.*

```
wkT : Γ \vdash_T A → Γ · y # B \vdash_C → Γ · y # B \vdash_T A
wkt : Γ \vdash_t t # A → Γ · y # B \vdash_C → Γ · y # B \vdash_t t # A
wks : Δ \vdash_S γ > Γ → Δ · y # B \vdash_C → Δ · y # B \vdash_S γ > Γ
```

Moreover, this theory satisfies inversion: Any sub-term of a derivable term is itself derivable. More precisely, we have the following (c.f. admitted results [13, Lemma 6] and [9, Lemma 6])

```
Γ\vdash_A→Γ\vdash : Γ \vdash_T A → Γ \vdash_C
Γ\vdash_t:A→Γ\vdash : Γ \vdash_t t # A → Γ \vdash_C
Δ\vdash_γ:Γ→Γ\vdash : Δ \vdash_S γ > Γ → Γ \vdash_C
Γ,x:A\vdash→Γ\vdash : Γ · x # A \vdash_C → Γ \vdash_C
Γ\vdash_t:A→Γ\vdash_A : Γ \vdash_t t # A → Γ \vdash_T A
Γ\vdash_s : Γ \vdash_T t ⇒[A] u → Γ \vdash_t t # A
Δ\vdash_γ:Γ→Δ\vdash : Δ \vdash_S γ > Γ → Δ \vdash_C
Γ\vdash_t : Γ \vdash_T t ⇒[A] u → Γ \vdash_t u # A
```

Structure of category with families. The study of the semantics relies on the structure of CwF of the theory. We can prove by induction on the derivation trees all the ingredients to show that GSeTT defines a CwF (at least set-theoretically). We do not show the final statement, as it requires developing a lot of category theory in Agda. The four following propositions show the structure of CwF of the theory GSeTT. We present them in the order in which we prove them, and they correspond to [9, Propositions 8, 9 and 10], where they are used without proof. We expect these results to be also necessary for a hypothetical internal notion of CwF.

► **Proposition 12.** *The action of a derivable substitution on a derivable type (resp. term) is again a derivable type (resp. term)*

$$\begin{aligned} []T &: \Gamma \vdash_T A \rightarrow \Delta \vdash_S \gamma > \Gamma \rightarrow \Delta \vdash_T (A \ [\gamma] T) \\ []t &: \Gamma \vdash_t t \# A \rightarrow \Delta \vdash_S \gamma > \Gamma \rightarrow \Delta \vdash_t (t \ [\gamma] t) \# (A \ [\gamma] T) \end{aligned}$$

► **Proposition 13.** *In the theory GSeTT, there is an identity substitution defined by*

$$\begin{aligned} Pre-id &: \forall (\Gamma : Pre-Ctx) \rightarrow Pre-Sub \\ Pre-id \ \emptyset &= \langle \rangle \\ Pre-id \ (\Gamma \cdot x \# A) &= \langle Pre-id \ \Gamma \ , \ x \mapsto Var \ x \rangle \end{aligned}$$

It acts trivially on types, terms and substitutions on a raw syntax level and it is derivable.

$$\begin{aligned} [id]T &: (A \ [\text{Pre-id } \Gamma] T) == A & \circ\text{-right-unit} &: (\gamma \circ \text{Pre-id } \Delta) == \gamma \\ [id]t &: (t \ [\text{Pre-id } \Gamma] t) == t & \Gamma \vdash_{id} \Gamma &: \Gamma \vdash_C \rightarrow \Gamma \vdash_S \text{Pre-id } \Gamma > \Gamma \end{aligned}$$

► **Proposition 14.** *The action of substitution is compatible with the composition of substitution*

$$\begin{aligned} [\circ]T &: \Gamma \vdash_T A \rightarrow \Delta \vdash_S \gamma > \Gamma \rightarrow \Theta \vdash_S \delta > \Delta \\ &\rightarrow ((A \ [\gamma] T) \ [\delta] T) == (A \ [\gamma \circ \delta] T) \\ [\circ]t &: \Gamma \vdash_t t \# A \rightarrow \Delta \vdash_S \gamma > \Gamma \rightarrow \Theta \vdash_S \delta > \Delta \\ &\rightarrow ((t \ [\gamma] t) \ [\delta] t) == (t \ [\gamma \circ \delta] t) \end{aligned}$$

► **Proposition 15.** *The composition of substitutions preserves the derivability of substitutions. Moreover, it is associative, and the identity is the left unit of the composition.*

$$\begin{aligned} \circ\text{-adm} &: \Delta \vdash_S \gamma > \Gamma \rightarrow \Theta \vdash_S \delta > \Delta \rightarrow \Theta \vdash_S (\gamma \circ \delta) > \Gamma \quad a : \Delta \vdash_S \gamma > \Gamma \\ &\rightarrow \Theta \vdash_S \delta > \Delta \rightarrow \Xi \vdash_S \theta > \Theta \rightarrow (\gamma \circ \delta) \circ \theta == \gamma \circ (\delta \circ \theta) \\ \circ\text{-left-unit} &: \Delta \vdash_S \gamma > \Gamma \rightarrow (\text{Pre-id } \Gamma \circ \gamma) == \gamma \end{aligned}$$

The result in Proposition 14 and 15 may not hold at the raw syntax level. For instance $id \ (\emptyset \cdot 0 \# * \cdot y \# *) \circ \langle x \mapsto x \ , \ y \mapsto y \rangle$ computes to $\langle x \mapsto x \ , \ y \mapsto y \rangle$.

4.3 Proof-theoretic considerations

We also prove notions that are more proof-theoretic about the theory. We also expect the uniqueness of derivation tree to be important for an hypothetical notion of internal CwF.

Decidability of type checking. In Martin-Löf type theory, we express that a type A is decidable, by exhibiting an inhabitant of the type $dec \ A = A + \neg A$.

► **Theorem 16.** *Type checking in the theory GSeTT is a decidable problem*

$$\begin{aligned} dec \vdash_C &: \forall \Gamma \rightarrow dec \ (\Gamma \vdash_C) & dec \vdash_t &: \forall \Gamma \ A \ t \rightarrow dec \ (\Gamma \vdash_t t \# A) \\ dec \vdash_T &: \forall \Gamma \ A \rightarrow dec \ (\Gamma \vdash_T A) & dec \vdash_S &: \forall \Delta \ \Gamma \ \gamma \rightarrow dec \ (\Delta \vdash_S \gamma > \Gamma) \end{aligned}$$

We prove this theorem by mutual induction of the derivation tree, however the structure of the induction is quite complicated and showing that it is well-founded is a hard problem. This is where the use of a proof assistant like `Agda` becomes extremely useful, since its termination checker is able to verify this automatically. Proving this theorem in `Agda` amounts to implementing a certified type checker for the theory `GSeTT`.

Uniqueness of derivation trees. We express uniqueness in the language of `HoTT`, by defining contractible types (i.e., types a unique inhabitant) and proposition types (i.e., types that are either empty or contractible).

```
is-contr A =  $\Sigma$  A ( $\lambda$  x  $\rightarrow$  ((y : A)  $\rightarrow$  x == y))
is-prop A =  $\forall$  (x y : A)  $\rightarrow$  is-contr (x == y)
```

► **Theorem 17.** *In the theory `GSeTT`, every derivable judgement has a unique derivation (stated without proof [9, Lemma 7])*

```
is-prop+C : is-prop ( $\Gamma \vdash_C$ )           is-prop+t : is-prop ( $\Gamma \vdash_t t \# A$ )
is-prop+T : is-prop ( $\Gamma \vdash_T A$ )       is-prop+S : is-prop ( $\Delta \vdash_S \gamma > \Gamma$ )
```

We again prove by mutual induction, and the proof is fairly straightforward. The absence of computation rule is key for this proof to be simple. Computation rules, that could be represented as type constructors performing homotopy coherent quotients would make such a result much more technical to prove. We can recover the typed syntax from the raw syntax and the judgements, by considering dependent pairs of an element of the raw syntax together with its judgement as follows. For instance for contexts, we define the type `Ctx = Σ Pre-Ctx (λ $\Gamma \rightarrow \Gamma \vdash_C$)`, and similarly for types, terms and substitutions, we define the types `Ty Γ` , `Tm Γ A` and `Sub $\Delta \Gamma$` .

4.4 Familial representability of types

We now define the disks and sphere contexts in `GSeTT`, which are families of contexts that play an important in the understanding of the semantics of the theory [9].

► **Definition 18.** *For every number n , we define a type $\Rightarrow_u n$ (u stands for “universal”) and two contexts `Pre-S n` and `Pre-D n` by mutual induction in the raw syntax as follows*

```
 $\Rightarrow_u 0 = *$ 
 $\Rightarrow_u (S n) = \text{Var } (2 n) \Rightarrow [ \Rightarrow_u n ] \text{Var } (2 n + 1)$ 
Pre-S 0 =  $\emptyset$ 
Pre-S (S n) = (Pre-D n)  $\cdot \ell$  (Pre-D n)  $\# \Rightarrow_u n$ 
Pre-D n = (Pre-S n)  $\cdot \ell$  (Pre-S n)  $\# \Rightarrow_u n$ 
```

► **Proposition 19.** *The disk and sphere contexts are valid contexts in the theory `GSeTT`, and the type \Rightarrow_u is derivable in the sphere context*

```
S $\vdash$  : Pre-S n  $\vdash_C$            D $\vdash$  : Pre-D n  $\vdash_C$            S $\vdash \Rightarrow$  : Pre-S n  $\vdash_T \Rightarrow_u n$ 
```

The sphere contexts play a particular role in the theory since they classify the types in a context: types in a context are equivalent to substitution from that context to a disk context. This is a result that we call familial representability of types [9], and that we formally prove in our foundational framework, using the definition of equivalence `is-equiv` usual to `HoTT`.

► **Theorem 20.** *For every context Γ , and any derivable substitution from Γ to a sphere, we define a derivable type in Γ by applying the substitution on the type $\Rightarrow_u _$. The resulting map defines an equivalence*

$$\begin{aligned} \text{Ty-n} &: \forall \Gamma \rightarrow \Sigma \mathbb{N} (\lambda n \rightarrow \text{Sub } \Gamma \text{ (Pre-S } n)) \rightarrow \text{Ty } \Gamma \\ \text{Ty-n } \Gamma \text{ (n , } (\gamma \text{ , } \Gamma \vdash \gamma : \text{Sn}) \text{)} &= ((\Rightarrow_u n) [\gamma] T) \text{ , } ([] T \text{ (S} \vdash \Rightarrow n) \Gamma \vdash \gamma : \text{Sn}) \end{aligned}$$

$$\text{Ty-classifier} : \forall \Gamma \rightarrow \text{is-equiv (Ty-n } \Gamma)$$

This result is substantially harder to prove formally than the previously mentioned ones, and relies on the uniqueness of derivation trees.

5 Formalisation and properties of the theory CaTT

In this section, we introduce our notion of globular type theory, and formalise it. We show that under suitable conditions, these theories satisfy the same good properties than GSeTT. We then construct CaTT and MCaTT as particular examples.

5.1 Globular type theories

Globular type theories are dependent type theories that have the same type structure as the theory GSeTT, but have term constructors. In order to describe not only the type theory CaTT, but also other dependent type theories, we define these term constructors generically. To this end, we assume a type I , which serves as an index to all the term constructors.

► **Definition 21.** *The raw syntax of a globular type theory is defined by the four mutually inductive types.*

```
-- Pre-Ctx, Pre-Ty and Pre-Sub defined like in GSeTT
data Pre-Tm where
  Var :  $\mathbb{N} \rightarrow \text{Pre-Tm}$ 
  Tm-c :  $\forall (i : I) \rightarrow \text{Pre-Sub} \rightarrow \text{Pre-Tm}$ 
```

► **Definition 22.** *The action of substitution on the raw syntax is computed the same way as the action of substitutions on the raw syntax of the theory GSeTT, except on terms where it is defined by*

```
-- [_ ] T and _o_ defined like in GSeTT
t [ <> ] t = t
Var x [ <  $\delta$  ,  $v \mapsto t$  > ] t = if  $x \equiv v$  then t else ((Var x) [  $\delta$  ] t)
Tm-c i  $\gamma$  [  $\delta$  ] t = Tm-c i ( $\gamma \circ \delta$ )
```

Inference rules of globular type theories. We give a presentation of a generic form for the introduction of the indexed term constructors in globular type theories. To achieve this, we parameterise the rules, in such a way that every term constructor corresponds to its own introduction rule. We allow to have term constructors in the pre-syntax that do not correspond to any derivable term, if the rule is inapplicable. From now on, we assume that the type I has decidable equality, that is, we have a term

$$\text{eqdecI} : \forall (x \ y : I) \rightarrow \text{dec (x == y)}$$

In order to parameterise the rules, we suppose that for every inhabitant i of the type I , there exists a context $Ci\ i$ in the raw syntax of `GSeTT` and a type $Ti\ i$ in the raw syntax of the globular type theory. Moreover, we assume that the context $Ci\ i$ is derivable in the theory `GSeTT`.

► **Definition 23.** *A globular type theory is a theory obtained from its syntax by imposing the same judgement rules as in the theory `GSeTT` for contexts, types and substitutions, and imposing for terms*

```
-- _⊢C_, _⊢T_ and _⊢S_>_ same as GSeTT
data _⊢t_#_ where
  var : Γ ⊢C → (x , A) ∈ Γ → Γ ⊢t (Var x) # A
  tm  : Ci i ⊢T Ti i → Δ ⊢S γ > Ci i → Δ ⊢t Tm-c i γ # (Ti i [ γ ]T)
```

Note that again, the judgements of the theory are defined mutually inductively, and this change propagates to the other types. Here we use an explicit equality $A == Ti\ i\ [\ \gamma]T$ instead of inlining the equality so that we have fine control over when it is eliminated, this allows us to stay compatible with avoiding the use of axiom `K`.

Properties of globular type theories. Most of the meta-theoretic properties extend from the theory `GSeTT` to any globular type theory, but there can be some difficulties in doing so.

► **Proposition 24.** *Every globular type theory satisfy all the results presented in Propositions 11, 12, 13, 14 and 15*

In this case, these results are a bit more involved to prove, because of the added dependency of terms on substitutions. Many results that could be proven separately in the case of `GSeTT` now depend on each other and have to be proven by mutual induction. Again, termination checking is not trivial, this is one instance where using `Agda` is a strong benefit.

► **Theorem 25** (c.f. Theorem 17). *In any globular type theory, every derivable judgement has a single derivation tree.*

Definition 18 of the disks and sphere contexts also makes sense in any globular type theory. We also call those the disk and sphere contexts in the raw syntax of the globular theory.

► **Theorem 26** (c.f. Proposition 19 and Theorem 20). *The disk and sphere context define valid contexts in any globular type theory, and the sphere contexts classify the types: There is an equivalence between the derivable types in a context and the substitutions from that context to a sphere context.*

Decidability of type checking. The decidability of type checking is a result that does not generalise as well to any globular type theory, because the generic form we have given for the rules is too permissive. Trying to reproduce the proof of `GSeTT` yields a proof whose termination cannot be checked by `Agda`: There is not a variant that decreases along the rules. And indeed, one may devise a globular type theory for which type checking is not decidable. But we can restrict our attention further, and consider theories satisfying an extra hypothesis

```
wfI : ∀ i → Ci i ⊢T Ti i → dimC (Ci i) ≤ dim (Ti i)
```

where `dim` is the dimension of a type (i.e., the number of iterated arrows it is built with) and `dimC` is the dimension of a context (i.e., the maximal dimension among the types it contains). The theory `CaTT` satisfies this hypothesis.

► **Theorem 27** (cf Theorem 16). *For every globular type theory satisfying the hypothesis wfI , the type checking is decidable.*

Proving this by induction is fairly straightforward, but ensuring that the induction is well-formed is quite involved. There is no obvious decreasing variant, so one needs to keep track of both the dimension and the number of nested term constructors in a precise way. This argument is non-trivial and the use of a termination checker like Agda's is extremely valuable.

5.2 Ps-contexts and the theory CaTT

We leverage the definition of globular type theory to formalise and prove some meta-theoretic properties of the theory CaTT. To this end, we define a particular type \mathbb{J} to index the term constructors, as well as the contexts Ci_j and the types Ti_j to define the inference rules.

Ps-contexts. In our formalism, there is no difference between the term constructors op and coh , both of them are of the form Tm-c . If anything, formally, op and coh correspond to families of term constructors and not term constructors. One of the ingredients to index these families are the ps-contexts that we formally define here.

► **Definition 28.** *We define the judgements $_ \vdash_{\text{ps}}$ and $_ \vdash_{\#}$ over the raw syntax of the type theory GSeTT as the following inductive types (where we denote $\mathbb{1}$ for $\ell \in \Gamma$)*

```
data _\vdash_{ps}_{\#}_ : Pre-Ctx → ℕ → Pre-Ty → Set where
  pss : (nil · 0 # *) \vdash_{ps} 0 # *
  psd : Γ \vdash_{ps} f # (Var x ⇒ [ A ] Var y) → Γ \vdash_{ps} y # A
  pse : Γ \vdash_{ps} x # A → ((Γ · ℓ # A) · S ℓ # Var x ⇒ [ A ] Var ℓ) \vdash_{ps}
    S ℓ # Var x ⇒ [ A ] Var ℓ
```

```
data _\vdash_{ps}_ : Pre-Ctx → Set where ps :
  Γ \vdash_{ps} x # * → Γ \vdash_{ps}
```

► **Proposition 29.** *The ps-contexts are valid contexts of the theory GSeTT.*

```
Γ \vdash_{ps} → Γ \vdash : Γ \vdash_{ps} → Γ \vdash
```

The relation \triangleleft . To work with ps-contexts formally, we define the relation \triangleleft introduced by Finster and Mimram [13]. The purpose of this relation is to perform inductive reasoning.

► **Definition 30.** *Given a context Γ , we define a generating relation $\Gamma, _ \triangleleft_0 _$, together with its transitive closure $\Gamma, _ \triangleleft _$ as follows*

```
data _,_<_0_ Γ x y : Set where
  <∂- : Γ \vdash_t (Var y) # (Var x ⇒ [ A ] Var z) → Γ, x <_0 y
  <∂+ : Γ \vdash_t (Var x) # (Var z ⇒ [ A ] Var y) → Γ, x <_0 y
```

```
data _,_<_ Γ x y : Set where
  gen : Γ, x <_0 y → Γ, x < y
  <T : Γ, x < z → Γ, z <_0 y → Γ, x < y
```

► **Proposition 31.** *The ps-contexts are linear for the relation $_, _ \triangleleft _$, i.e., whenever Γ is a ps-context, the relation $\Gamma, _ \triangleleft _$ defines a linear order on the variables of Γ .*

```
ps-<-linear : ∀ Γ → Γ \vdash_{ps} → <-linear Γ
```

The proof of this proposition in [13] relies on semantic consideration and the link between GSeTT and globular sets. In our approach, we instead give a purely syntactic proof of this result. This makes it very technical. The main ingredient of the proof is a subtle invariant, stating that whenever we have $\Gamma \vdash_{\text{ps}} \mathbf{x} \# \mathbf{A}$ and $\Gamma, \mathbf{x} \triangleleft \mathbf{y}$, then necessarily \mathbf{y} is an iterated target of \mathbf{x} in Γ .

► **Theorem 32.** *The judgement \vdash_{ps} is decidable, and any two derivation of the same judgement of this form are equal.*

$$\text{is-prop} \vdash_{\text{ps}} : \forall \Gamma \rightarrow \text{is-prop} (\Gamma \vdash_{\text{ps}}) \quad \text{dec} \vdash_{\text{ps}} : \forall \Gamma \rightarrow \text{dec} (\Gamma \vdash_{\text{ps}})$$

These results are proven by induction on the derivation trees, but they are not straightforward. For instance in the case of the uniqueness, a derivation of $\Gamma \vdash_{\text{ps}}$ necessarily comes from a derivation of the form $\Gamma \vdash_{\text{ps}} \mathbf{x} \# *$ and by induction this derivation is necessarily unique. But the hard part is to prove that there can only be a unique \mathbf{x} such that $\Gamma \vdash_{\text{ps}} \mathbf{x} \# *$. Using the \triangleleft -linearity, we can prove a more general lemma: if we have a $\Gamma \vdash_{\text{ps}} \mathbf{x} \# \mathbf{A}$ and $\Gamma \vdash_{\text{ps}} \mathbf{y} \# \mathbf{A}$ with \mathbf{A} and \mathbf{B} two types of the same dimension, then $\mathbf{x} = \mathbf{y}$. The decidability also presents a difficulty: The rule `psd` contains variables in its premises that are not bound in its conclusion. However, these variables belong to the context, so we can solve this issue by enumeration of the variables and \triangleleft -linearity.

Index of term constructors. With the ps-contexts, we can define the index type for the term constructors in the theory CaTT. Recall that the term constructors in this theory are defined by `opΓ,A` and `cohΓ,A`, where Γ is a ps-context and A is a type. In our informal presentation, we also required the side conditions (C_{op}) and (C_{coh}) in the derivation rules. For convenience, we integrate these side conditions in the index type in our formalisation, and define \mathbf{J} to be the type of pairs of the form (Γ, A) , where Γ is a ps-context and A is a type satisfying either (C_{op}) or (C_{coh}) . The conditions (C_{op}) and (C_{coh}) are a bit subtle to formalise in HoTT, because one variable may appear several times in the same term, so there may be several witnesses that a term contains all the desired variables. But the intended semantics is propositional. To solve this issue, we propositionally truncate the type witnessing that a variable appears in type. We realise the truncation by using the type `set` of sets of numbers, for which membership is a proposition. We define the type $\mathbf{A} \subset \mathbf{B}$ of witnesses that a set \mathbf{A} is included in a set \mathbf{B} , as well as the type $\mathbf{A} \stackrel{\circ}{=} \mathbf{B} = (\mathbf{A} \subset \mathbf{B}) \times (\mathbf{B} \subset \mathbf{A})$ of set equality. These two types are propositions since we are only manipulating finite subsets of \mathbb{N} .

► **Definition 33.** *We define the sets of source and target variables of a ps-context. First we define the i -sources and i -targets by induction on the judgement $\Gamma \vdash_{\text{ps}} \mathbf{x} \# \mathbf{A}$ as lists.*

$$\begin{aligned} \text{src}_i\text{-var } i \text{ pss} &= \text{if } i \equiv 0 \text{ then nil else (nil :: 0)} \\ \text{src}_i\text{-var } i \text{ (psd } \Gamma \vdash_{\text{ps}} \mathbf{x}) &= \text{src}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} \\ \text{src}_i\text{-var } i \text{ (pse } \Gamma = \Gamma \text{ } A = A \Gamma \vdash_{\text{ps}} \mathbf{x}) &\text{ with } \text{dec} \leq i \text{ (S (dim A))} \\ \dots \mid \text{inl } i \leq \text{SdimA} &= \text{src}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} \\ \dots \mid \text{inr } \text{SdimA} < i &= (\text{src}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} :: (\ell \Gamma)) :: (\text{S } (\ell \Gamma)) \end{aligned}$$

$$\begin{aligned} \text{tgt}_i\text{-var } i \text{ pss} &= \text{if } i \equiv 0 \text{ then nil else (nil :: 0)} \\ \text{tgt}_i\text{-var } i \text{ (psd } \Gamma \vdash_{\text{ps}} \mathbf{x}) &= \text{tgt}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} \\ \text{tgt}_i\text{-var } i \text{ (pse } \Gamma = \Gamma \text{ } A = A \Gamma \vdash_{\text{ps}} \mathbf{x}) &\text{ with } \text{dec} \leq i \text{ (S (dim A))} \\ \dots \mid \text{inl } i \leq \text{SdimA} &= \text{if } i \equiv \text{S (dim A)} \text{ then drop}(\text{tgt}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x}) :: (\ell \Gamma) \\ &\quad \text{else } \text{tgt}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} \\ \dots \mid \text{inr } \text{SdimA} < i &= (\text{tgt}_i\text{-var } i \Gamma \vdash_{\text{ps}} \mathbf{x} :: (\ell \Gamma)) :: (\text{S } (\ell \Gamma)) \end{aligned}$$

2:18 Formalisation of Dependent Type Theory: The Example of CaTT

Here the `with` construction matches on the decidability of the order in \mathbb{N} . For instances matching on `dec-≤ i n` produces two cases of type `i ≤ n` and `n < i`. Moreover `drop` takes a list and removes its head. The source and target sets of a `ps-context` are the sets corresponding to the source and target lists in maximal dimension.

$$\begin{aligned} \text{src-var } (\Gamma, \text{ps } \Gamma \vdash \text{ps} x) &= \text{set-of-list } (\text{src}_i\text{-var } (\text{dimC } \Gamma) \Gamma \vdash \text{ps} x) \\ \text{tgt-var } (\Gamma, \text{ps } \Gamma \vdash \text{ps} x) &= \text{set-of-list } (\text{tgt}_i\text{-var } (\text{dimC } \Gamma) \Gamma \vdash \text{ps} x) \end{aligned}$$

► **Definition 34.** We define the type `_is-full-in_`, witnessing whether either the condition (C_{op}) or (C_{coh}) is satisfied, as follows

`data _is-full-in_ where`

$$\begin{aligned} \text{Cop} &: (\text{src-var } \Gamma) \stackrel{o}{=} ((\text{varT } A) \cup \text{-set } (\text{vart } t)) \\ &\rightarrow (\text{tgt-var } \Gamma) \stackrel{o}{=} ((\text{varT } A) \cup \text{-set } (\text{vart } u)) \rightarrow (t \Rightarrow [A] u) \text{ is-full-in } \Gamma \\ \text{Ccoh} &: (\text{varC } (\text{fst } \Gamma)) \stackrel{o}{=} (\text{varT } A) \rightarrow A \text{ is-full-in } \Gamma \end{aligned}$$

where `varT` (resp. `vart`, `varC`) is the set of variables associated to a type (resp. to a term, to a context).

► **Definition 35.** The type $J = \Sigma (\text{ps-ctx} \times \text{Ty}) \lambda \{(\Gamma, A) \rightarrow A \text{ is-full-in } \Gamma\}$ is the index type for the term constructors of the theory `CaTT`. It is the types of pairs (Γ, A) where Γ is a `ps-context` and A is a raw type satisfying (C_{op}) or (C_{coh}) .

The raw syntax of `CaTT` is the raw syntax of the globular type theory indexed by J .

The type theory CaTT. We define the dependent type theory `CaTT` by adding rules to the raw syntax. It suffices to define a context `Ci i` and a type `Ti i` for every i of type J .

► **Definition 36.** Considering a term $(((\Gamma, \Gamma \vdash \text{ps}), A), A\text{-full})$ of type J , we pose

$$\begin{aligned} \text{Ci } (((\Gamma, \Gamma \vdash \text{ps}), A), A\text{-full}) &= (\Gamma, \Gamma \vdash \text{ps} \rightarrow \Gamma \vdash \Gamma \vdash \text{ps}) \\ \text{Ti } (((\Gamma, \Gamma \vdash \text{ps}), A), A\text{-full}) &= A \end{aligned}$$

The theory `CaTT` is the globular type theory obtained from these assignments.

► **Proposition 37.** J has decidable equality and satisfies the well-foundedness condition

$$\begin{aligned} \text{eqdecJ} &: \forall (x y : J) \rightarrow \text{dec } (x == y) \\ \text{wfJ} &: \forall j \rightarrow \text{Ci } j \vdash_T \text{Ti } j \rightarrow \text{dimC } (\text{Ci } j) \leq \text{dim } (\text{Ti } j) \end{aligned}$$

Since this definition realises `CaTT` as a particular case of a globular type theory, it enjoys all the properties that we have already proved for them. In particular we have already proved

► **Theorem 38.** In the theory `CaTT` the following statements hold.

- The theory support weakening and derivability is preserved by the inference rules.
- The theory `CaTT` defines a category with families.
- Every derivable judgement in `CaTT` has a unique derivation tree.
- The sphere contexts in `CaTT` classify the types.
- Type checking is decidable in the theory `CaTT`.

5.3 The theory M CaTT

In addition to CaTT , we have also defined a dependent type theory to describe monoidal weak ω -categories, that we call MCaTT . There are actually two slightly different but equivalent formulations for this theory [8] and [7], and only the second one is a globular type theory. Thanks to the generic indexing mechanism that we have provided, we were able to formalise this theory as well, straightforwardly following the presentation given in [7]. Our understanding of the semantics relies on translations back and forth between CaTT and MCaTT . Such translations are defined by induction on the syntax, formalising them and proving their correctness is a technical challenge that we leave for future works.

6 Conclusion and further work

We have presented a full formalisation of the foundational aspects of the dependent type theory CaTT . Although this dependent type theory is quite simple, in that it does not have computation rules, proving formally all the relevant aspects that we expected turned out to be a substantial amount of work with highly non-trivial challenges to solve. In particular for some of the aspects such as the decidability of type checking, the use of a proof assistant such as *Agda* appears almost mandatory given the subtlety of the arguments. The notion of globular type theory is a limited attempt at a framework to carry out this work once in a generic enough setting that it can be used for CaTT and MCaTT , and it shows how a careful indexing of term constructors allows for some genericity while still being able to retain enough precision to show meaningful properties. It would be valuable to connect this notion with the work of Leena-Subramaniam [20], and a reasonable conjecture is that it corresponds to finitary monads on globular sets. More general approaches are being developed by Bauer, Haselwarter and Lumsdaine [6] and their development with Petkovic of the proof assistant *Andromeda*, Umeura [26] and Gylterud with the Myott project⁷ [15]. Ultimately it would be valuable to have a formalised definitive framework in which all those meta-theoretic properties are proved once and for all.

The formalisation that we have presented, and in particular the proof for the decidability of type checking constitutes a verified implementation of a type checker for the theory CaTT . We have developed regular implementation of such a type checker⁸ in *OCaml*. However, to improve the user experience, we have defined some meta-operations (called suspension and functorialization) on the syntax of the theory, that we proved correct manually [10, 7]. To avoid relying on the correctness of our implementation, the software computes the result of these operations and checks them like any user inputs. This leads to inefficiency in the implementation, and is not very satisfying. A better practice would be to define and prove formally those meta-operations, and then export the results to executable code in order to have a natively certified, but computationally light implementation of these meta-operations.

The work we have presented also shows once again the connection between the problem of internalising higher structures and internalising dependent type theory in HoTT . The meet point is the coherence issue, and we have avoided it here by only focusing solely on the syntax. A natural, but much harder follow up for this work would be to formalise the semantics of this theory. There is already some progress made in this direction by Uemura [23] with the definition of $\infty\text{-CwF}$, and we conjecture that internalizing this notion in HoTT would allow

⁷ <https://git.app.uib.no/Hakon.Gylterud/myott>

⁸ <https://github.com/thibautbenjamin/catt>

us to define an internal notion of models of CaTT, which could provide a suitable definition of weak ω -categories internally to HoTT. In general, giving a formulation of higher categorical results in terms of a syntax and a dependent type theory allows to perform some amount of reasoning that can be formalised within a proof assistant. We believe that it constitutes an asset for higher category theory, where the complexity of the theory itself quickly becomes a meaningful obstacle for any non-trivial exploration by hand of the theories.

References

- 1 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.
- 2 Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Extending homotopy type theory with strict equality. *CoRR*, 2016. arXiv:1604.03799.
- 3 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. *ACM SIGPLAN Notices*, 51(1):18–29, 2016. doi:10.1145/2837614.2837638.
- 4 Thorsten Altenkirch and Ondrej Rypacek. A syntactical approach to weak omega-groupoids. In *26th Computer Science Logic (CSL'12)*, volume 16 of *LIPICs*. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2012. doi:10.4230/LIPICs.CSL.2012.16.
- 5 Michael A Batanin. Monoidal globular categories as a natural environment for the theory of weak n-categories. *Advances in Mathematics*, 136(1):39–103, 1998. doi:10.1006/aima.1998.1724.
- 6 Andrej Bauer, Philipp G Haselwarter, and Peter LeFanu Lumsdaine. A general definition of dependent type theories. *CoRR*, 2020. arXiv:2009.05539.
- 7 Thibaut Benjamin. *A type theoretic approach to weak ω -categories and related higher structures*. PhD thesis, Institut Polytechnique de Paris, 2020.
- 8 Thibaut Benjamin. Monoidal weak ω -categories as models of a type theory. *CoRR*, abs/2111.14208, 2021. arXiv:2111.14208.
- 9 Thibaut Benjamin, Eric Finster, and Samuel Mimram. Globular weak ω -categories as models of a type theory. *CoRR*, 2021. arXiv:2106.04475.
- 10 Thibaut Benjamin and Samuel Mimram. Suspension et Functorialité: Deux Opérations Implicites Utiles en CaTT. In *Journées Francophones des Langages Applicatifs*, 2019.
- 11 James Chapman. Type theory should eat itself. *Electronic notes in theoretical computer science*, 228:21–36, 2009. doi:10.1016/j.entcs.2008.12.114.
- 12 Peter Dybjer. Internal Type Theory. In *Types for Proofs and Programs. TYPES 1995*, pages 120–134. Springer, Berlin, Heidelberg, 1996. doi:10.1007/3-540-61780-9_66.
- 13 Eric Finster and Samuel Mimram. A Type-Theoretical Definition of Weak ω -Categories. In *2017 32nd Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–12, 2017. doi:10.1109/LICS.2017.8005124.
- 14 Alexander Grothendieck. Pursuing stacks. Unpublished manuscript, 1983.
- 15 Hakon Gylderud. Defining and relating theories. Presentation HoTT Electronic Seminar, 2021.
- 16 Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, 25(5):1116–1131, 2015. doi:10.1017/S0960129514000528.
- 17 Ambrus Kaposi, András Kovács, and Ambroise Lafont. For finitary induction-induction, induction is enough. In *TYPES 2019*, volume 175 of *LIPICs*, pages 6:1–6:30. Schloss Dagstuhl-Leibniz-Zentrum für Informatik GmbH, 2020. doi:10.4230/LIPICs.TYPES.2019.6.
- 18 Nicolai Kraus. Internal ∞ -categorical models of dependent type theory: Towards 2l_{tt} eating hott. In *36th Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–14. IEEE, 2021. doi:10.1109/LICS52264.2021.9470667.
- 19 Ambroise Lafont, Tom Hirschowitz, and Nicolas Tabareau. Types are weak omega-groupoids, in Coq. *TYPES 2018*, 2018.
- 20 Chaitanya Leena-Subramaniam. *From dependent type theory to higher algebraic structures*. PhD thesis, Université Paris 7, 2021. arXiv:2110.02804.

- 21 Peter LeFanu Lumsdaine. Weak ω -categories from intensional type theory. In *TLCA*, pages 172–187. Springer, 2009. doi:10.1007/978-3-642-02273-9_14.
- 22 Georges Maltsiniotis. Grothendieck ∞ -groupoids, and still another definition of ∞ -categories. *CoRR*, 2010. arXiv:1009.2331.
- 23 Hoang Kim Nguyen and Taichi Uemura. ∞ -type theories. In *abstract presented at the online workshop HoTT/UF'20*, 2020.
- 24 Mike Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow), 2014. URL: <https://homotopytypetheory.org/2014/03/03/hott-should-eat-itself/>.
- 25 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.
- 26 Taichi Uemura. A general framework for the semantics of type theory. *CoRR*, 2019. arXiv:1904.04097.
- 27 Benno Van Den Berg and Richard Garner. Types are weak ω -groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011. doi:10.1112/plms/pdq026.

Strictification of Weakly Stable Type-Theoretic Structures Using Generic Contexts

Rafaël Bocquet  

Eötvös Loránd University, Budapest, Hungary

Abstract

We present a new strictification method for type-theoretic structures that are only weakly stable under substitution. Given weakly stable structures over some model of type theory, we construct equivalent strictly stable structures by evaluating the weakly stable structures at generic contexts. These generic contexts are specified using the categorical notion of familial representability. This generalizes the local universes method of Lumsdaine and Warren.

We show that generic contexts can also be constructed in any category with families which is freely generated by collections of types and terms, without any definitional equality. This relies on the fact that they support first-order unification. These free models can only be equipped with weak type-theoretic structures, whose computation rules are given by typal equalities. Our main result is that any model of type theory with weakly stable weak type-theoretic structures admits an equivalent model with strictly stable weak type-theoretic structures.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, strictification, coherence, familial representability, unification

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.3

1 Introduction

Type-theoretic structures are usually required to be strictly stable under substitution. However many structures arising from category theory and homotopy theory are only specified up to isomorphism, equivalence or homotopy. They are then only *weakly* stable under substitution. This is for instance the case for the identity types arising from weak factorization systems [1] and for the constructive simplicial model of Gambino and Henry [12]. In order to interpret type theories into such structures, we have to use *strictification theorems* that replace weakly stable structures by strictly stable ones.

Generally, a strictification method is a procedure that constructs, given an input model with weakly stable type structures, another model with stable type structures, connected to the original model via a zigzag of equivalences (for a suitable notion of equivalence). Several strictification methods are known [14, 9, 22, 8, 2], with different constraints on the type theories and models. We recall two of the most general constructions.

Right adjoint splitting: A strictification method [14, 9] due to Hofmann defines a new model \mathcal{C}_* in which types over a context Γ are coherent families of types of the base model \mathcal{C} , indexed by the substitutions $\Delta \rightarrow \Gamma$. This is a *cofree* construction: we pack together all the data that is needed when substituting, along with witnesses that this data is coherent, i.e. that different ways of substituting coincide, up to isomorphism or equivalence.

This method is known to work for extensional type theories, i.e. type theories with the equality reflection rule, but it does not directly apply to most models arising from homotopy theory. In presence of equality reflection it is sufficient to consider families of types that are coherent up to isomorphism. A generalization would need to consider homotopy-coherent families of types and terms, that include coherence conditions in all dimensions. Defining a workable notion of homotopy-coherent family is however not easy. We note that coherence theorems proven in Uemura’s PhD thesis [27] essentially involve such homotopy-coherent families.



© Rafaël Bocquet;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 3; pp. 3:1–3:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Left adjoint splitting/local universes: The local universes method [22] of Lumsdaine and Warren generalizes Voevodsky’s use of universes to obtain stability in the simplicial model [21]. It instantiates the weakly stable structures at suitable *generic contexts*. Strict stability under substitution then follows from the stability of the construction of the generic contexts. In order to ensure the existence of the generic contexts, this strictification method replaces the base model \mathcal{C} by a new model, the *local universes model* \mathcal{C}_1 , also called the *left adjoint splitting*, in which types over Γ are replaced by triples (V, E, χ) , where (V, E) is a local universe, consisting of a closed context V and of a type E over V , and χ is a substitution from Γ to V . The generic contexts of the type and term formers then only depend on the local universes of the type parameters, but not on the map χ nor on the term parameters; this ensures that they are invariant under substitution. The construction of these generic contexts requires the existence of some local exponentials in the underlying category of the base model. This condition is called condition (LF).

Generic contexts

We present a new general strictification method. Like the local universes method, our method instantiates the weakly stable structures at generic contexts. In the local universes construction, the generic contexts can only depend on the shapes of types, but not on the structure of terms. We give a finer characterization of the (universal) properties required by the generic contexts, using the categorical notion of *familial representability* [4, 5].

If x is an element over a context Γ of a presheaf X (such as the presheaf of types or the presheaf of terms of a given type), a generalization of x is an element x_0 over some context Γ_0 , along with a substitution $\rho : \Gamma \rightarrow \Gamma_0$ such that $x = x_0[\rho]$. A most general generalization is a terminal object in the category of generalizations. When they exist, the most general generalizations of x and $x[\sigma]$ coincide (at least up to isomorphism). The presheaf X is *familially representable* if all of its elements admit most general generalizations (with some additional naturality condition). Equivalently, a presheaf is familially representable when it is a coproduct of a family of representable presheaves.

A weakly stable type-theoretic operation (type or term former) T on a category \mathcal{C} is given by a dependent non-natural transformation $T : \forall(\Gamma : \mathbf{Ob}_{\mathcal{C}})(x : X_{\Gamma}) \rightarrow Y_{\Gamma}(x)$, where X is a presheaf over \mathcal{C} and Y is a dependent presheaf over X . When the presheaf X is familially representable, we can define a natural transformation $T^s : \forall(\Gamma : \mathcal{C}^{\text{op}})(x : X_{\Gamma}) \rightarrow Y_{\Gamma}(x)$ by $T^s(\Gamma, x) \triangleq T(\Gamma_0, x_0)[\rho]$, where $x_0 : X_{\Gamma_0}$ is the most general generalization of x . Here we have defined a strictly stable operation T^s as the instantiation of the weakly stable operation T at the *generic context* Γ_0 .

The presheaves X that may occur as the sources of type-theoretic operations all have a specific shape: they are given by *polynomial sorts*, which are obtained by closing the basic sorts (types and terms) under dependent products (with arities in terms) and dependent sums. They correspond to the objects of the representable map category [26] that encodes the type theory. We say that a model (a category with families) has **familially representable polynomial sorts** when the presheaves of elements of polynomial sorts are all familially representable. Any weakly stable type-theoretic structure over a base model that satisfies that condition can be replaced by a stable type-theoretic structure.

We obtain the following theorem.

► **Theorem 1.** *Let \mathcal{C} be a CwF equipped with weakly stable identity types. If \mathcal{C} has familially representable polynomial sorts, then \mathcal{C} can be equipped with stable identity types that are equivalent to the weakly stable identity types.*

It is straightforward to generalize this construction to any other weakly stable type-theoretic structure.

The condition (LF) of the local universe method [22] implies that the local universe model \mathcal{C}_l has familially representable polynomial sorts; thus the local universe method factors through our method.

Free categories with families

There are models that have familially representable polynomials sorts without satisfying condition (LF). We show that this is the case for all categories with families (CwFs) that are freely generated by some collection of generating types and terms. Freely generated CwFs can also be seen as generalized (i.e. dependently sorted) algebraic theories [6] without equations. Using the terminology of weak factorization systems, the freely generated CwFs can be described as the cellular objects with respect to some set I of CwF morphisms.

Thanks to the absence of equations, free CwFs support *first-order unification*; any two unifiable types, terms or substitutions admit a *most general unifier*. These most general unifiers are used to construct most general generalization for polynomial sorts.

► **Theorem 2.** *If a CwF \mathcal{C} is freely generated (I -cellular), then it has locally familially representable polynomials sorts.*

Strictification of weakly stable weak type-theoretic structures

By the small object argument, every CwF \mathcal{C} admits an I -cellular replacement, which is a freely generated CwF \mathcal{C}_0 equipped with a *trivial fibration* $F : \mathcal{C}_0 \rightarrow \mathcal{C}$. A trivial fibration is a morphism that is surjective on types and terms; in particular it is a kind of equivalence between CwFs. Thus every CwF \mathcal{C} admits an equivalent CwF \mathcal{C}_0 that has familially representable polynomial sorts. Furthermore all type and term formers can be lifted from \mathcal{C} to \mathcal{C}_0 along F , except that definitional equalities cannot be lifted.

In other words, every *weak* type-theoretic structure can be lifted. A weak type-theoretic structure is a type-theoretic structure that is presented without definitional equalities. Typically, their computation rules are specified up to typal equality, rather than up to definitional equality. For example, weak identity types (under the name of propositional identity types) were introduced in [28]. The computation rule of the weak J eliminator is only given by a typal equality $J\beta : \text{Id}(J(d, x, \text{refl}), d)$. Similarly, we can consider weak Π -types, weak Σ -types, etc.

We thus have two ways to weaken the usual presentation of a type-theoretic structure: we can weaken either the stability under substitution and/or the computation rules. In general we may want to compare weakly stable, weakly computational structures with strictly stable, strictly computational structures. As it is hard to do this comparison directly, it has to be split into multiple steps. The present paper provides comparisons between weakly stable, weak and strictly stable, weak structures. There is ongoing work [3] by the author towards coherence theorems that compare strictly stable, weak structures with strictly stable, strict structures.

Combining the previous results, we obtain the following theorem:

► **Theorem 3.** *Let \mathcal{C} be a CwF with weakly stable weak identity types. Then there exists a CwF \mathcal{D} with stable weak identity types and a trivial fibration $F : \mathcal{D} \rightarrow \mathcal{C}$ in \mathbf{CwF} that weakly preserves identity types.*

This theorem can straightforwardly be extended to any other weakly stable weak type-theoretic structure.

In general, we are interested in coherence theorems that are more powerful than Theorem 3. We expect that Theorem 3 can be part of the proofs of such coherence theorems; this is discussed in Section 6.

2 Background

We work in a constructive metatheory.

2.1 Presheaf categories

We use the internal language of the category $\mathbf{Psh}(\mathcal{C})$ of presheaves over a base category \mathcal{C} ; any presheaf category is a model of extensional type theory [15]. This justifies the use of higher-order abstract syntax (HOAS) to describe type-theoretic structures over a base category \mathcal{C} .

If $\Gamma : \mathbf{Ob}_{\mathcal{C}}$ is an object of \mathcal{C} , the corresponding representable presheaf is written $y(\Gamma)$. A morphism $f : \Gamma \rightarrow \Delta$ can be identified with the natural transformation $f : y(\Gamma) \rightarrow y(\Delta)$.

If X is a presheaf, we identify global elements of the exponential presheaf $(y(\Gamma) \rightarrow X)$ with elements of the evaluation of X at Γ . If $x : y(\Gamma) \rightarrow X$ and $f : \Delta \rightarrow \Gamma$, we may write $x[f]$ for the restriction of x along f .

We write $\int_{\mathcal{C}} X$ for the category of elements of X ; its objects are pairs (Γ, x) with $x : y(\Gamma) \rightarrow X$, and a morphism $(\Delta, x') \rightarrow (\Gamma, x)$ is a morphism $\rho : \Delta \rightarrow \Gamma$ such that $x' = x[\rho]$.

A dependent presheaf over X is a presheaf over $\int_{\mathcal{C}} X$. If Y is a dependent presheaf over X and $x : y(\Gamma) \rightarrow X$, global elements of the presheaf $(\gamma : y(\Gamma)) \rightarrow Y(x(\gamma))$ coincide with elements of the evaluation of Y at Γ and x .

The presheaf universe classifying the i -small dependent presheaves is denoted by \mathcal{U}_i ; we will generally omit the universe level i . Dependent products are written $(a : A) \rightarrow B(a)$, sometimes with a leading \forall quantifier. Dependent sums are written $(a : A) \times B(a)$. The terminal presheaf is denoted by \top .

If $x : y(\Gamma) \rightarrow X$ and $y : (\gamma : y(\Gamma)) \rightarrow Y(x(\gamma))$, we write $\langle x, y \rangle$ for the corresponding element of $(\gamma : y(\Gamma)) \rightarrow (a : X(\gamma)) \times (b : Y(a))$. We write $\langle \rangle$ for the unique element of $y(\Gamma) \rightarrow \top$.

2.2 Categories with Families

We use categories with families [10, 7] as our models of type theory. We recall how the notion of local representability, which encodes the context extensions, is derived from the (non-local) notion of representability. We will similarly derive a notion of local familial representability from the notion of familial representability in Section 3.1.

► **Definition 4.** A dependent presheaf $Y : X \rightarrow \mathcal{U}$ is **locally representable** if for every element $x : y(\Gamma) \rightarrow X$, the restricted presheaf

$$\begin{aligned} Y|_x & : \mathbf{Psh}(\mathcal{C}/\Gamma) \\ Y|_x(\rho : \Delta \rightarrow \Gamma) & \triangleq Y(x[\rho] : y(\Delta) \rightarrow X) \end{aligned}$$

is representable. ┘

► **Definition 5.** A *family* over a category \mathcal{C} is a pair $(\mathbf{T}\mathbf{y}, \mathbf{T}\mathbf{m})$ consisting of a presheaf $\mathbf{T}\mathbf{y} : \mathcal{U}$ and of a dependent presheaf $\mathbf{T}\mathbf{m} : \mathbf{T}\mathbf{y} \rightarrow \mathcal{U}$. We say that the family has **representable elements** when $\mathbf{T}\mathbf{m}$ is locally representable. \lrcorner

► **Definition 6.** A *category with families* (CwF) is a category \mathcal{C} equipped with a terminal object \diamond , along with a global family $(\mathbf{T}\mathbf{y}_{\mathcal{C}}, \mathbf{T}\mathbf{m}_{\mathcal{C}})$ with representable elements. \lrcorner

The local representability condition describes the context extensions. If $\Gamma : \mathbf{Ob}_{\mathcal{C}}$ and $A : \mathbf{y}(\Gamma) \rightarrow \mathbf{T}\mathbf{y}_{\mathcal{C}}$, we have an extended context $\Gamma.A : \mathbf{Ob}_{\mathcal{C}}$ and a natural isomorphism $\mathbf{y}(\Gamma.A) \simeq (\gamma : \mathbf{y}(\Gamma)) \times (a : \mathbf{T}\mathbf{m}_{\mathcal{C}}(A(\gamma)))$. We will often identify the two sides of this isomorphism. The two projections out of this dependent sum are the projection morphism $\mathbf{p}_A : \Gamma.A \rightarrow \Gamma$ and the variable term $\mathbf{q}_A : ((\gamma, a) : \mathbf{y}(\Gamma.A)) \rightarrow \mathbf{T}\mathbf{m}_{\mathcal{C}}(A(\gamma))$. If $\rho : \Delta \rightarrow \Gamma$, we write ρ^+ for the canonical morphism $\rho^+ : \Delta.A[\rho] \rightarrow \Gamma.A$, i.e. $\rho^+ = \langle \rho \circ \mathbf{p}_A, \mathbf{q}_A \rangle$.

We write **CwF** for the 1-category of CwFs and strict CwF morphisms.

We write $(\mathbf{T}\mathbf{y}^*, \mathbf{T}\mathbf{m}^*)$ for the family of **telescopes** of a family $(\mathbf{T}\mathbf{y}, \mathbf{T}\mathbf{m})$. It is defined as the following inductive-recursive family, internally to $\mathbf{Psh}(\mathcal{C})$:

$$\begin{aligned} \mathbf{T}\mathbf{y}^* & : \mathcal{U} \\ \mathbf{T}\mathbf{m}^* & : \mathbf{T}\mathbf{y}^* \rightarrow \mathcal{U} \\ \diamond & : \mathbf{T}\mathbf{y}^* \\ \mathbf{T}\mathbf{m}^*(\diamond) & \triangleq \top \\ _ _ & : (\Delta : \mathbf{T}\mathbf{y}^*)(A : \mathbf{T}\mathbf{m}^*(\Delta) \rightarrow \mathbf{T}\mathbf{y}) \rightarrow \mathbf{T}\mathbf{y}^* \\ \mathbf{T}\mathbf{m}^*(\Delta.A) & \triangleq (\delta : \mathbf{T}\mathbf{m}^*(\Delta)) \times (a : \mathbf{T}\mathbf{m}(A(\delta))) \end{aligned}$$

In other words, a telescope of types $A : \mathbf{T}\mathbf{y}^*$ is a finite sequence $A_1.A_2.\dots.A_n$ of dependent types. A telescope of terms $a : \mathbf{T}\mathbf{m}^*(A)$ is a sequence $a_1 : A_1, a_2 : A_2(a_1), \dots, a_n : A_n(a_1, a_2, \dots)$ of terms. If $(\mathbf{T}\mathbf{y}, \mathbf{T}\mathbf{m})$ has representable elements, then so does $(\mathbf{T}\mathbf{y}^*, \mathbf{T}\mathbf{m}^*)$; the context extensions of $(\mathbf{T}\mathbf{y}^*, \mathbf{T}\mathbf{m}^*)$ are iterations of the context extensions of $(\mathbf{T}\mathbf{y}, \mathbf{T}\mathbf{m})$.

There is a canonical map $\mathbf{T}\mathbf{y}_{\mathcal{C}}^* \rightarrow \mathbf{Ob}_{\mathcal{C}}$ sending any closed telescope to the corresponding extension of the empty context. We say that \mathcal{C} is **contextual** when that map is bijective. In that case, we identify the objects of \mathcal{C} and the closed telescopes. Up to that identification, the Yoneda embedding $\mathbf{y} : \mathbf{Ob}_{\mathcal{C}} \rightarrow \mathcal{U}$ coincides with the restriction of $\mathbf{T}\mathbf{m}_{\mathcal{C}}^* : \mathbf{T}\mathbf{y}_{\mathcal{C}}^* \rightarrow \mathcal{U}$ to closed telescopes.

► **Definition 7.** If \mathcal{C} is a contextual CwF, we characterize its variables by an inductive family $\mathbf{Var} : (\Gamma : \mathbf{Ob}_{\Gamma})(A : \mathbf{y}(\Gamma) \rightarrow \mathbf{T}\mathbf{y}_{\mathcal{C}})(a : \forall \gamma \rightarrow \mathbf{T}\mathbf{m}_{\mathcal{C}}(A(\gamma))) \rightarrow \mathbf{Set}$, generated by:

$$\frac{}{\mathbf{Var}_{\Gamma.A,A[\mathbf{p}_A]}(\mathbf{q}_A)} \quad \frac{\mathbf{Var}_{\Gamma,A}(x)}{\mathbf{Var}_{\Gamma.B,A[\mathbf{p}_B]}(x[\mathbf{p}_B])}$$

2.3 Strictly stable and weakly stable weak identity types

We give definitions of the structures of stable and weakly stable weak identity types using the internal language of $\mathbf{Psh}(\mathcal{C})$. Note that the weakly stable structures cannot be fully be specified internally; it involves an external quantification over contexts.

We use Paulin-Mohring's variant of the identity type elimination principle, as it is better behaved than Martin-Löf's eliminator in the absence of other type-theoretic structures. In the absence of Π -types, Martin-Löf's eliminator needs to be parametrized by an additional telescope, as introduced by Gambino and Garner [11]. This is discussed in more details in [23, 18, 3].

3:6 Strictification of Weakly Stable Type-Theoretic Structures Using Generic Contexts

Paulin-Mohring's eliminator corresponds to based path induction, in which the left endpoint of a path is fixed.

$$\frac{A \text{ type} \quad x : A}{[y : A] \text{ Id}(A, x, y) \text{ type}} \quad \frac{A \text{ type} \quad x : A}{\text{refl}(A, x) : \text{Id}(A, x, x)}$$

$$\frac{A \text{ type} \quad x : A \quad [y : A, p : \text{Id}(A, x, y)] P(y, p) \text{ type} \quad d : P(x, \text{refl}(A, x))}{[y : A, p : \text{Id}(A, x, y)] J(A, x, P, d, y, p) : P(y, p)}$$

We consider *weak* identity types, which means that their computation rule is given by a typal equality, rather than a definitional equality.

$$\frac{A \text{ type} \quad x : A \quad [y : A, p : \text{Id}(A, x, y)] P(y, p) \text{ type} \quad d : P(x, \text{refl}(A, x))}{J\beta(A, x, P, d, y, p) : \text{Id}(P(x, \text{refl}(A, x)), J(A, x, P, d, x, \text{refl}(A, x)), d)}$$

Note that the type former Id has two parameters (A and x) and one index y . The fact that y is an index cannot be seen in the definition of the stable type-former Id as a natural transformation $\text{Id} : (A : \text{Ty}_{\mathcal{C}})(x, y : \text{Tm}_{\mathcal{C}}(A)) \rightarrow \text{Ty}_{\mathcal{C}}$. However it changes the definition of the weakly stable type-former Id ; we will have a type $\text{Id}_{\Gamma, A, x}$ in the extended context $\Gamma.(y : A)$.

► **Definition 8.** A (strictly stable) *weak identity type structure* on a family (Ty, Tm) consists of an *introduction structure*

$$\begin{aligned} \text{Id} & : \forall(A : \text{Ty})(x, y : \text{Tm}(A)) \rightarrow \text{Ty}, \\ \text{refl} & : \forall A \ x \rightarrow \text{Tm}(\text{Id}(A, x, x)), \end{aligned}$$

along with a *weak elimination structure*

$$\begin{aligned} J & : \forall(A : \text{Ty})(x : \text{Tm}(A)) \\ & \quad (P : \forall(y : \text{Tm}(A))(p : \text{Tm}(\text{Id}(A, x, y))) \rightarrow \text{Ty}) \\ & \quad (d : \text{Tm}(P(x, \text{refl}(A, x)))) \\ & \quad \rightarrow \forall y \ p \rightarrow \text{Tm}(P(y, p)), \\ J\beta & : \forall(A : \text{Ty})(x : \text{Tm}(A)) \\ & \quad (P : \forall(y : \text{Tm}(A))(p : \text{Tm}(\text{Id}(A, x, y))) \rightarrow \text{Ty}) \\ & \quad (d : \text{Tm}(P(x, \text{refl}(A, x)))) \\ & \quad \rightarrow \text{Tm}(\text{Id}(P(x, \text{refl}(A, x)), J(A, x, P, d, x, \text{refl}(A, x)), d)). \quad \lrcorner \end{aligned}$$

We also define the weakly stable weak identity types.

► **Definition 9.** A *Id-introduction context* is a triple (Γ, A, x) , where

$$\begin{aligned} \Gamma & : \text{Ob}_{\mathcal{C}}, \\ A & : y(\Gamma) \rightarrow \text{Ty}, \\ x & : (\gamma : y(\Gamma)) \rightarrow \text{Tm}(A(\gamma)). \end{aligned}$$

Here Γ is an object of \mathcal{C} , and A and x are types and terms that only depend on Γ .

A **weakly stable identity type introduction structure** consists, for every **ld-introduction context** (Γ, A, x) , of operations

$$\begin{aligned} \text{ld}_{(\Gamma, A, x)} &: \forall(\gamma : \mathfrak{y}(\Gamma))(y : \mathbf{Tm}(A(\gamma))) \rightarrow \mathbf{T}\mathfrak{y}, \\ \text{refl}_{(\Gamma, A, x)} &: \forall(\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathbf{Tm}(\text{ld}_{(\Gamma, A, x)}(\gamma, x(\gamma))). \end{aligned}$$

A **ld-elimination context** over an **ld-introduction context** (Γ, A, x) is a tuple (Δ, γ, P, d) , where

$$\begin{aligned} \Delta &: \text{Ob}_{\mathcal{C}}, \\ \gamma &: \Delta \rightarrow \Gamma, \\ P &: \forall(\delta : \mathfrak{y}(\Delta))(y : \mathbf{Tm}(A(\gamma(\delta))))(p : \mathbf{Tm}(\text{ld}_{(\Gamma, A, x)}(\gamma(\delta), y))) \rightarrow \mathbf{T}\mathfrak{y}, \\ d &: \forall(\delta : \mathfrak{y}(\Delta)) \rightarrow \mathbf{Tm}(P(\delta, x(\gamma(\delta))), \text{refl}_{(\Gamma, A, x)}(\gamma(\delta))). \end{aligned}$$

A **weakly stable identity type elimination structure** consists, for every **ld-elimination context** (Δ, γ, P, d) over (Γ, A, x) , of operations

$$\begin{aligned} \mathbf{J}_{(\Gamma, A, x, \Delta, \gamma, P, d)} &: \forall(\delta : \mathfrak{y}(\Delta))(y : \mathbf{Tm}(A(\gamma(\delta))))(p : \mathbf{Tm}(\text{ld}_{(\Gamma, A, x)}(\gamma(\delta), y))) \rightarrow \mathbf{Tm}(P(\delta, y, p)), \\ \mathbf{J}\beta_{(\Gamma, A, x, \Delta, \gamma, P, d)} &: \forall(\delta : \mathfrak{y}(\Delta)) \rightarrow \text{ld}_{(\Delta, P', d)}(\delta, \mathbf{J}_{(\Gamma, A, x, \Delta, \gamma, P, d)}(\delta, x(\gamma(\delta)), \text{refl}_{(\Gamma, A, x)}(\gamma(\delta))), \\ P'(\delta') &\triangleq P(\delta', x(\gamma(\delta')), \text{refl}_{\Gamma}(\gamma(\delta'))). \quad \lrcorner \end{aligned}$$

Note that strictly stable identity types are weakly stable identity types satisfying additional naturality conditions. In presence of weakly stable weak identity types, we have well-behaved notions of contractible types, type equivalences, etc.

► **Proposition 10.** *The weakly stable weak identity types are indeed weakly stable: for every ld-introduction context (Γ, A, x) and substitution $\rho : \Delta \rightarrow \Gamma$, the canonical map*

$$\mathbf{Tm}(\text{ld}_{(\Delta, A[\rho], x[\rho])}) \rightarrow \mathbf{Tm}(\text{ld}_{(\Gamma, A, x)}[\rho])$$

is an equivalence over $\Delta.A[\rho]$. ◀

► **Definition 11.** *A CwF morphism $F : \mathcal{C} \rightarrow \mathcal{D}$ weakly preserves weakly stable weak identity types if for every ld-introduction context (Γ, A, x) of \mathcal{C} , then the canonical map*

$$\mathbf{Tm}_{\mathcal{D}}(\text{ld}_{(F(\Gamma), F(A), F(x))}) \rightarrow \mathbf{Tm}_{\mathcal{D}}(F(\text{ld}_{(\Gamma, A, x)}))$$

is an equivalence over $F(\Gamma.A)$. ◀

2.4 Trivial fibrations and freely generated CwFs

We recall the definition of the (cofibrations, trivial fibrations) weak factorization system on **CwF**. The same weak factorization system on the category **CwA** of Categories with Attributes, which is equivalent to **CwF**, was introduced by Kapulkin and Lumsdaine [19, Definition 4.12].

► **Definition 12.** *A morphism $F : \mathcal{C} \rightarrow \mathcal{D}$ of CwFs is a **trivial fibration** if its actions on types and terms are surjective, i.e. if it satisfies the following lifting conditions:*

(type lifting) *For every object $\Gamma : \text{Ob}_{\mathcal{C}}$ and type $A : \mathfrak{y}(F(\Gamma)) \rightarrow \mathbf{T}\mathfrak{y}_{\mathcal{D}}$, there exists a type*

$$A_0 : \mathfrak{y}(\Gamma) \rightarrow \mathbf{T}\mathfrak{y}_{\mathcal{C}} \text{ such that } F(A_0) = A.$$

(term lifting) *For every object $\Gamma : \text{Ob}_{\mathcal{C}}$, type $A : \mathfrak{y}(\Gamma) \rightarrow \mathbf{T}\mathfrak{y}_{\mathcal{C}}$ and term $a : (\gamma : \mathfrak{y}(F(\Gamma))) \rightarrow$*

$$\mathbf{Tm}_{\mathcal{D}}(F(A)(\gamma)), \text{ there exists a term } a_0 : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathbf{Tm}_{\mathcal{C}}(A(\gamma)) \text{ such that } F(a_0) = a,$$

where the existential quantifications are strong, meaning that F is equipped with a choice of lifts. ◻

The (cofibrations, trivial fibrations) weak factorization system on \mathbf{CwF} is cofibrantly generated by the set $I = \{I^{\text{ty}}, I^{\text{tm}}\}$, where

$$I^{\text{ty}} : \text{Free}(\Gamma : \text{Ob}) \rightarrow \text{Free}(\mathbf{A} : \mathfrak{y}\Gamma \rightarrow \text{Ty}),$$

$$I^{\text{tm}} : \text{Free}(\mathbf{A} : \mathfrak{y}\Gamma \rightarrow \text{Ty}) \rightarrow \text{Free}(\mathbf{a} : (\gamma : \mathfrak{y}\Gamma) \rightarrow \text{Tm}(\mathbf{A}(\gamma))).$$

Here $\text{Free}(\Gamma : \text{Ob})$ is the CwF freely generated by an object Γ , $\text{Free}(\mathbf{A} : \mathfrak{y}\Gamma \rightarrow \text{Ty})$ is the CwF freely generated by an object Γ and a type \mathbf{A} over Γ , and $\text{Free}(\mathbf{a} : (\gamma : \mathfrak{y}\Gamma) \rightarrow \text{Tm}(\mathbf{A}(\gamma)))$ is the CwF freely generated by Γ , \mathbf{A} and a term \mathbf{a} of type \mathbf{A} over Γ .

We also recall the definition of I -cellular maps and objects in \mathbf{CwF} .

► **Definition 13.** A *basic I -cellular map* $\mathcal{C} \rightarrow \mathcal{D}$ is a pushout of a coproduct of maps in I ; it freely adjoins to a model \mathcal{C} a collection of new types and terms whose contexts and types are from \mathcal{C} . An *I -cellular map* is a sequential composition of a sequence $(\iota_i^{i+1} : \mathcal{C}_i \rightarrow \mathcal{C}_{i+1})_{i \leq \omega}$ of basic I -cellular maps.

A CwF \mathcal{C} is an *I -cellular object* (or *I -cell complex*) if the unique map $\mathbf{0} \rightarrow \mathcal{C}$ is an I -cellular map. ◻

By the small object argument, every morphism of CwFs can be factored as an I -cellular map followed by a trivial fibration. In particular, for any CwF \mathcal{C} , the factorization of the unique map $\mathbf{0} \rightarrow \mathcal{C}$ provides an I -cellular object \mathcal{C}_0 and a trivial fibration $\mathcal{C}_0 \rightarrow \mathcal{C}$.

► **Proposition 14.** If $F : \mathcal{C} \rightarrow \mathcal{D}$ is a trivial fibration between CwFs and \mathcal{D} is equipped with weakly stable weak identity types, then \mathcal{C} can be equipped with weakly stable weak identity types that are strictly preserved by F .

Proof. By lifting each component of the weakly stable weak identity types of \mathcal{D} . ◀

► **Proposition 15.** Any I -cellular CwF is contextual.

Proof. Let \mathbb{N} be the terminal contextual CwFs; its contexts are natural numbers, and it has a unique type and a unique term over every context. A CwF \mathcal{C} is contextual if and only there exists a unique CwF morphism $\mathcal{C} \rightarrow \mathbb{N}$; such a morphism gives the length of every context of \mathcal{C} .

Now take an I -cellular CwF \mathcal{C} . For any other CwF \mathcal{D} , a CwF morphism $\mathcal{C} \rightarrow \mathcal{D}$ is determined by the image of the generating types and terms of \mathcal{C} . Since \mathbb{N} has a unique type and a unique term, there exists a unique CwF morphism $\mathcal{C} \rightarrow \mathbb{N}$, sending each generating type or term to the unique type or term of \mathbb{N} . Thus \mathcal{C} is contextual, as needed. ◀

The collections of generating types and terms of an I -cellular CwF \mathcal{C} can be obtained from the decomposition of $\mathbf{0} \rightarrow \mathcal{C}$ as an I -cellular map. We use a (**red, bold**) font to distinguish the generating types and terms from arbitrary types and terms.

► **Construction 16.** Let \mathcal{C} be an I -cellular CwF. Then we construct sets $\text{GenTy}_{\mathcal{C}} : \text{Set}$ of *generating types* and $\text{GenTm}_{\mathcal{C}} : \text{Set}$ of *generating terms* such that

- For every $\mathbf{S} : \text{GenTy}_{\mathcal{C}}$, we have an object $\partial\mathbf{S} : \text{Ob}_{\mathcal{C}}$ and a dependent type $\mathbf{S} : \partial\mathbf{S} \rightarrow \text{Ty}_{\mathcal{C}}$.
- For every $\mathbf{f} : \text{GenTm}_{\mathcal{C}}$, we have an object $\partial\mathbf{f} : \text{Ob}_{\mathcal{C}}$, a type $T\mathbf{f} : \partial\mathbf{f} \rightarrow \text{Ty}_{\mathcal{C}}$ and a dependent term $\mathbf{f} : \forall(\tau : \partial\mathbf{f}) \rightarrow \text{Tm}_{\mathcal{C}}(T\mathbf{f}(\tau))$.

The components $\partial\mathbf{S}$ and $\partial\mathbf{f}$ specify the dependencies (or the boundary) of the generating types and terms. The component $T\mathbf{f}$ gives the output type of a generating term.

Details of Construction 16. See Appendix A. ◀

We can obtain an syntactic description of the general types and terms of an I -cellular CwF as the well-typed trees built out of the generating types and terms.

► **Construction 17.** *Given an object $\Gamma : \text{Ob}_{\mathcal{C}}$, we define inductive families of sets*

$$\begin{aligned} \text{NfTy} &: \forall \Delta \ (y(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}) \rightarrow \text{Set}, \\ \text{Nf}_{\Gamma}^* &: \forall \Delta \ (y(\Gamma) \rightarrow \text{Tm}_{\mathcal{C}}^*(\Delta)) \rightarrow \text{Set}, \\ \text{Nf}_{\Gamma} &: \forall A \ (y(\Gamma) \rightarrow \text{Tm}_{\mathcal{C}}(A)) \rightarrow \text{Set}, \end{aligned}$$

generated by the following (unnamed) constructors:

$$\begin{array}{c} \frac{\mathbf{S} : \text{GenTm}_{\mathcal{C}} \quad \text{Nf}_{\Gamma}^*(\tau)}{\text{NfTy}_{\Gamma}(\mathbf{S}[\tau])} \\ \\ \frac{}{\text{Nf}_{\Gamma}^*(\langle \rangle)} \quad \frac{\text{Nf}_{\Gamma}^*(\delta) \quad \text{Nf}_{\Gamma}(a)}{\text{Nf}_{\Gamma}^*(\langle \delta, a \rangle)} \\ \\ \frac{\text{Var}_{\Gamma}(a)}{\text{Nf}_{\Gamma}(a)} \quad \frac{\mathbf{f} : \text{GenTm}_{\mathcal{C}} \quad \text{Nf}_{\Gamma}^*(\tau)}{\text{Nf}_{\Gamma}(\mathbf{f}[\tau])} \end{array}$$

Then for every type A , substitution σ or term a , there is a unique element of $\text{NfTy}(A)$, $\text{Nf}^*(\sigma)$ or $\text{Nf}(a)$. In other words, types, terms and telescopes of terms admit a unique normal form. ┘

Details of Construction 17. See Appendix A. ◀

3 Generic contexts

3.1 Familiably representable presheaves

We recall the notion of *familiably representable* presheaf [4, 5].

► **Definition 18.** *Let \mathcal{C} be a category and $X : \mathcal{U}$ be a presheaf over \mathcal{C} .*

The following conditions are equivalent:

1. *Every connected component of the category of elements $\int_{\mathcal{C}} X$ is equipped with a terminal object. If $x : y(\Gamma) \rightarrow X$ is an element, the terminal object $x_0 : y(\Gamma_0) \rightarrow X$ of its connected component is called the **most general generalization** of x .*
2. *The presheaf X can be decomposed as a coproduct of representable presheaves*

$$X \simeq \coprod_{i:I} (y(X_i))$$

for some family of objects $X : I \rightarrow \text{Ob}_{\mathcal{C}}$ indexed by some set I .

3. *For every element $x : y(\Gamma) \rightarrow X$, we have an element $x_0 : y(\Gamma_0) \rightarrow X$ and there is a unique morphism $f : \Gamma \rightarrow \Gamma_0$ such that $x = x_0[f]$. Furthermore, x_0 depends strictly naturally on Γ .*

*When they hold, we say that X is **familiably representable**.* ┘

Proof. See [4] for the equivalence between conditions (1) and (2). Condition (3) is an unfolding of condition (1). ◀

► **Definition 19.** A dependent presheaf $Y : X \rightarrow \mathcal{U}$ is **locally familially representable** if for every element $x : y(\Gamma) \rightarrow X$, the restricted presheaf

$$Y|_x \quad : \quad \mathbf{Psh}(\mathcal{C}/\Gamma)$$

$$Y|_x(\rho : \Delta \rightarrow \Gamma) \triangleq Y(x[\rho] : y(\Delta) \rightarrow X)$$

is familially representable. ┘

Unfolding the definition, a dependent presheaf $Y : X \rightarrow \mathcal{U}$ is locally familially representable if for every element $x : y(\Delta) \rightarrow X$, morphism $\rho : \Gamma \rightarrow \Delta$ and element $y : (\gamma : y(\Gamma)) \rightarrow Y(x(\rho(\gamma)))$, there is, strictly naturally in Γ , a map $\rho_0 : \Gamma_0 \rightarrow \Delta$ and an element $y : (\gamma : y(\Gamma_0)) \rightarrow Y(x(\rho_0(\gamma)))$ such that there is a unique map $f : \Gamma \rightarrow \Gamma_0$ satisfying $\rho = \rho_0 \circ f$ and $y = y_0[f]$. The object Γ_0 can be seen as the extension of the context Δ that classifies the connected component of y .

► **Proposition 20.** If a family $Y : X \rightarrow \mathcal{U}$ is locally familially representable, the family of telescopes $Y^* : X^* \rightarrow \mathcal{U}$ is also locally familially representable. ◀

3.2 Polynomial sorts

► **Definition 21.** Let \mathcal{C} be a *CwF*. We define global families $\mathbf{BSort}_{\mathcal{C}}$ of **basic sorts**, $\mathbf{MonoSort}_{\mathcal{C}}$ of **monomial sorts** and $\mathbf{PolySort}_{\mathcal{C}}$ of **polynomial sorts**. We write $\mathbf{Elem}(-)$ for the elements of these families. Note that they do not necessarily have representable elements.

■ A **basic sort** is either $\mathbb{t}y$ or $\mathbb{t}m(A)$ for some $A : \mathbf{Ty}_{\mathcal{C}}$.

$$\mathbf{Elem}(\mathbb{t}y) \quad \triangleq \quad \mathbf{Ty}_{\mathcal{C}}$$

$$\mathbf{Elem}(\mathbb{t}m(A)) \triangleq \mathbf{Tm}_{\mathcal{C}}(A)$$

We can view the basic sorts $\mathbb{t}y$ and $\mathbb{t}m(-)$ as codes for the presheaves of types and terms.

■ A **monomial sort** $[\Delta \vdash A]$ (or $[\delta : \Delta \vdash A(\delta)]$) consists of a telescope $\Delta : \mathbf{Ty}_{\mathcal{C}}^*$ and a dependent basic sort $A : \mathbf{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \mathbf{BSort}_{\mathcal{C}}$. It represents dependent functions from Δ to A , or equivalently elements of A in a context extended by Δ .

$$\mathbf{Elem}([\Delta \vdash A]) \triangleq (\delta : \mathbf{Tm}_{\mathcal{C}}^*(\Delta)) \rightarrow \mathbf{Tm}_{\mathcal{C}}(A(\delta))$$

■ A **polynomial sort** is a telescope of monomial sorts:

$$\mathbf{PolySort}_{\mathcal{C}} \triangleq \mathbf{MonoSort}_{\mathcal{C}}^*. \quad \text{┘}$$

Thus a polynomial sort is a dependent sum of dependent products of basic sorts. Since dependent sums distribute over dependent products, $\mathbf{PolySort}_{\mathcal{C}}$ is closed under dependent products with arities in $\mathbf{Tm}_{\mathcal{C}}$.

The parameters of (both weakly and strictly stable) type-theoretic structures are all described by (closed) polynomial sorts. For instance, the parameters of an **ld-introduction** structure are given by the closed polynomial sort

$$\partial \mathbf{ld} \triangleq (A : \mathbb{t}y) \times (x : \mathbb{t}m(A)).$$

The parameters of an **ld-elimination** structure are specified by the polynomial sort

$$\partial \mathbf{J} \triangleq ((A, x) : \partial \mathbf{ld}) \times (P : [y : A(\gamma), p : \mathbf{ld}_{\Gamma}(\gamma, y) \vdash \mathbb{t}y]) \times (d : \mathbb{t}m(P(x(\gamma), \mathbf{refl}_{\Gamma}(\gamma))))).$$

► **Definition 22.** We say that a *CwF* \mathcal{C} has **familially representable polynomial sorts** if for every closed polynomial sort P , the presheaf $\mathbf{Elem}(P)$ is familially representable. ┘

3.3 Strictification

► **Theorem 1.** *Let \mathcal{C} be a CwF equipped with weakly stable identity types. If \mathcal{C} has familially representable polynomial sorts, then \mathcal{C} can be equipped with stable identity types that are equivalent to the weakly stable identity types.*

Proof. The proof works for identity types with either a weak or a strict computation rule.

We first consider the closed polynomial $\partial \text{Id} \triangleq (A : \mathbb{t}y) \times (x : \mathbb{t}m(A))$.

Let $\langle A, x \rangle : y(\Gamma) \rightarrow \text{Elem}(\partial \text{Id})$ be the parameters of the stable Id -introduction structure over a context Γ . Since \mathcal{C} has generic polynomial contexts, we can find a most general generalization $\langle A_0, x_0 \rangle : y(\Gamma_0) \rightarrow \text{Elem}(\partial \text{Id})$ of $\langle A, x \rangle$. By the universal property of $\langle A_0, x_0 \rangle$, we have a map $f : \Gamma \rightarrow \Gamma_0$ such that $\langle A, x \rangle = \langle A_0, x_0 \rangle[f]$.

We then pose

$$\begin{aligned} \text{Id}_\Gamma^s(A, x, y) &\triangleq \text{Id}_{(\Gamma_0, A_0, x_0)}[\langle f, y \rangle], \\ \text{refl}_\Gamma^s(A, x) &\triangleq \text{refl}_{(\Gamma_0, A_0, x_0)}[f]. \end{aligned}$$

Since most general generalizations are strictly natural, $(\text{Id}^s, \text{refl}^s)$ is a stable Id -introduction structure.

Now consider the polynomial sort

$$\partial J \triangleq ((A, x) : \partial \text{Id}) \times (P : [y : A, p : \text{Id}^s(A, x, y)] \mathbb{t}y) \times (d : \mathbb{t}m(P(x, \text{refl}^s(A, x))))$$

Let $\langle A, x, P, d \rangle : (\gamma : y(\Gamma)) \rightarrow \text{Elem}(\partial J)$ be the parameters of the stable Id -elimination structure over Γ , A and x . Since \mathcal{C} has generic polynomial contexts, we can find a most general generalization $\langle A_1, x_1, P_1, d_1 \rangle : y(\Gamma_1) \rightarrow \partial J$. There is a unique map $g : \Gamma \rightarrow \Gamma_1$ such that $\langle A, x, P, d \rangle = \langle A_1, x_1, P_1, d_1 \rangle[g]$.

We can also obtain the most general generalization $\langle A_0, x_0 \rangle : y(\Gamma_0) \rightarrow \partial \text{Id}$ of $\langle A_1, x_1 \rangle$. We have a map $f : \Gamma_1 \rightarrow \Gamma_0$ such that $\langle A_1, x_1 \rangle = \langle A_0, x_0 \rangle[f]$. By the universal property of most general generalizations, $\langle A_0, x_0 \rangle$ is also the most general generalization of $\langle A, x \rangle$. Thus by definition of Id^s , we have $\text{Id}_\Gamma^s(A, x, y) = \text{Id}_{(\Gamma_0, A_0, x_0)}[\langle f \circ g, y \rangle]$.

We can finally pose

$$\begin{aligned} J_\Gamma^s(A, x, P, d, y, p) &\triangleq J_{(\Gamma_0, A_0, x_0, \Gamma_1, f, P_1, d_1)}[\langle g, y, p \rangle], \\ J\beta_\Gamma^s(A, x, P, d) &\triangleq J\beta_{(\Gamma_0, A_0, x_0, \Gamma_1, f, P_1, d_1)}[g]. \end{aligned}$$

This determines a stable Id -elimination structure $(J^s, J\beta^s)$. Note that if $J\beta$ is strict, then $J\beta^s$ is also strict.

By Proposition 10 the stable Id -types are equivalent to the weakly stable identity types. ◀

3.4 The local universes method

We show that the local universes strictification method [22] factors through ours.

► **Definition 23** ([22, Definition 3.1.3]). *A CwF \mathcal{C} satisfies the condition (LF) if its underlying category has finite products, and given maps $Z \xrightarrow{g} Y \xrightarrow{f} X$, if f is a display map and g is either a display map or a product projection, then a dependent exponential $\Pi[f, g]$ exists. ◻*

In the above definition, a display map is a finite composite of projections maps $\mathbf{p}_A : \Gamma.A \rightarrow \Gamma$; equivalently a display map is a projection map $\mathbf{p}_\Delta : \Gamma.\Delta \rightarrow \Gamma$ where Γ is an object of \mathcal{C} and Δ is a telescope over Γ .

3:12 Strictification of Weakly Stable Type-Theoretic Structures Using Generic Contexts

Condition (LF) can essentially be unfolded into the following two representability conditions:

- For every object $\Gamma : \text{Ob}_{\mathcal{C}}$, telescope $\Delta : \mathfrak{y}(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}^*$ and object $\Theta : \text{Ob}_{\mathcal{C}}$, the presheaf

$$(\gamma : \mathfrak{y}(\Gamma)) \times (\text{Tm}_{\mathcal{C}}^*(\Delta(\gamma)) \rightarrow \mathfrak{y}(\Theta))$$

is representable.

- For every object $\Gamma : \text{Ob}_{\mathcal{C}}$, telescope $\Delta : \mathfrak{y}(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}^*$ and type

$$A : (\gamma : \mathfrak{y}(\Gamma))(\delta : \text{Tm}_{\mathcal{C}}^*(\Delta(\gamma))) \rightarrow \text{Ty}_{\mathcal{C}},$$

the presheaf

$$(\gamma : \mathfrak{y}(\Gamma)) \times ((\delta : \text{Tm}_{\mathcal{C}}^*(\Delta(\gamma))) \rightarrow \text{Tm}_{\mathcal{C}}(A(\gamma, \delta)))$$

is representable.

► **Definition 24.** Let \mathcal{C} be a CwF.

A **local universe** is a pair (V, E) , where $V : \text{Ob}_{\mathcal{C}}$ is an object of \mathcal{C} and $E : \mathfrak{y}(V) \rightarrow \text{Ty}_{\mathcal{C}}$ is a type over V .

The **local universe model** $\mathcal{C}_!$ is another CwF over the same base category. We write $(\text{Ty}_!, \text{Tm}_!)$ for its family of types and terms.

A type of $\mathcal{C}_!$ is a triple (V, E, χ) , where (V, E) is a local universe, and $\chi : \mathfrak{y}(V)$. There is a natural transformation $\text{Ty}_! \rightarrow \text{Ty}_{\mathcal{C}}$, sending (V, E, χ) to $E(\chi)$.

The terms of $\mathcal{C}_!$ are induced by this natural transformation: $\text{Tm}_!(V, E, \chi) \triangleq \text{Tm}_{\mathcal{C}}(E(\chi))$. The local representability of the dependent presheaf $\text{Tm}_!$ follows from the local representability of $\text{Tm}_{\mathcal{C}}$. ┘

There is a CwF morphism $\mathcal{C}_! \rightarrow \mathcal{C}$ lying over the identity functor. That morphism is surjective on types and bijective on terms. In particular, it is a trivial fibration.

Any weakly stable type-theoretic structure can be lifted along $\mathcal{C}_! \rightarrow \mathcal{C}$. Since $\mathcal{C}_! \rightarrow \mathcal{C}$ is injective on terms, definitional equalities between terms can also be lifted. It is however not generally possible to lift definitional equalities between types.

► **Proposition 25.** If \mathcal{C} satisfies condition (LF), then $\mathcal{C}_!$ has familially representable polynomial sorts.

Proof. See Appendix A. ◀

4 Most general generalizations in free CwFs

In this section we prove the following result.

► **Theorem 2.** If a CwF \mathcal{C} is freely generated (*I*-cellular), then it has locally familially representable polynomial sorts.

We fix an *I*-cellular CwF \mathcal{C} . We use the explicit description of the types and terms of \mathcal{C} that was given in Construction 16.

4.1 First-order unification

First-order unification [24, 13] is usually presented for free untyped or simply typed theories, but it is folklore that the same unification procedure is also valid for free dependently typed theories¹, i.e. for freely generated CwFs. In our setting, this means that the category of cones over any pair of parallel substitutions is either empty or has a terminal object, which is then the *most general unifier* of the two substitutions.

We prove a slightly stronger result, for contexts that are split into *flexible* and *rigid* parts. The unification procedure can only change the flexible part.

► **Definition 26.** An **unification context** is an object of the form $\Gamma.\Delta$, where Δ is a telescope over Γ . The variables of Γ are called **flexible variables**, while the variables from Δ are called **rigid variables**.

A morphism of unification contexts is a substitution that preserves the rigid variables, i.e. a substitution of the form $\rho^+ : \Theta.\Delta[\rho] \rightarrow \Gamma.\Delta$ for some $\rho : \Theta \rightarrow \Gamma$. ◻

► **Definition 27 (Unifiers).** Let $\Gamma.\Delta$ be a unification context and X be a dependent presheaf over $\mathfrak{y}(\Gamma.\Delta)$. A **unifier** of a pair $a, b : (x : \mathfrak{y}(\Gamma.\Delta)) \rightarrow X(x)$ of parallel elements of X is a morphism $\rho : \Theta \rightarrow \Gamma$ such that $a[\rho^+] = b[\rho^+]$. We say that a and b are **unifiable** if there merely exists a unifier.

A **most general unifier** is a terminal unifier. ◻

► **Lemma 28 (Instantiation).** Let Γ be a context, $a : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$ be a variable from Γ and $b : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$ be a term of type A such that $b \neq a$.

If the terms a and b are unifiable, then we can construct a most general unifier $\Gamma[a := b]$. Moreover, the length of $\Gamma[a := b]$ is less than the length of Γ .

Proof. We have a bijective renaming $\Gamma \simeq (\gamma_0 : \Gamma_0).\Gamma_1(\gamma_0)$ where Γ_0 is the support of the term b . Up to this renaming, we have $A : \mathfrak{y}(\Gamma_0) \rightarrow \mathsf{T}_{\mathcal{Y}_{\mathcal{C}}}$, $b : (\gamma_0 : \mathfrak{y}(\Gamma_0)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma_0))$ and $a : (\gamma_0 : \mathfrak{y}(\Gamma_0))(\gamma_1 : \mathsf{Tm}_{\mathcal{C}}^*(\Gamma_1(\gamma_0))) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma_0))$.

The variable a cannot belong to the support Γ_0 of b , since a and b are unifiable and different; this is the *occurs check* of first-order unification. Indeed, assuming that a did belong to Γ_0 and considering the unifier ρ of a and b , the term $b[\rho]$ would be infinite.

Thus a is a variable from Γ_1 and we can write $\Gamma_1(\gamma_0) = (\gamma_2 : \Gamma_2(\gamma_0)).(a : A(\gamma_0)).\Gamma_3(\gamma_0, \gamma_2, a)$.

We now pose $\Gamma[a := b] \triangleq (\gamma_0 : \Gamma_0).(\gamma_2 : \Gamma_2(\gamma_0)).\Gamma_3(\gamma_0, \gamma_2, b)$. It is the most general unifier of a and b . ◀

► **Lemma 29 (Strengthening).** Let $\Gamma.\Delta$ be a unification context, $a : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$ be a term over Γ and $b : ((\gamma, \delta) : \mathfrak{y}(\Gamma.\Delta)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$ be a term of type $A[\mathfrak{p}_{\Delta}]$.

If the terms $a[\mathfrak{p}_{\Delta}]$ and b are unifiable, then there exists a (necessarily unique) term $b' : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$ such that $b = b'[\mathfrak{p}_{\Delta}]$.

Proof. Let $\rho : \Omega \rightarrow \Gamma$ be a unifier of $a[\mathfrak{p}_{\Delta}]$ and b . Then $b[\rho^+] = a[\mathfrak{p}_{\Delta}][\rho^+] = a[\rho][\mathfrak{p}_{\Delta}[\rho]]$. Thus $b[\rho^+]$ cannot depend on any variable from $\Delta[\rho]$. Since ρ^+ preserves the variables of Δ , the term b cannot depend on any variable from Δ . Therefore it can be strengthened to some term $b' : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathsf{Tm}_{\mathcal{C}}(A(\gamma))$. ◀

¹ This is observed by Simon Henry in <https://mathoverflow.net/questions/307373/on-a-surprising-property-of-free-theories>.

► **Theorem 30** (First-order unification). *Let $\Gamma.\Delta$ be a unification context and $X : y(\Gamma.\Delta) \rightarrow \mathcal{U}$ a dependent presheaf of the form $\text{Tm}_{\mathcal{C}}^*(\Xi)$, $\text{Ty}_{\mathcal{C}}$ or $\text{Tm}_{\mathcal{C}}(A(-))$. If there exists a unifier $\sigma : \Theta \rightarrow \Gamma$ of a pair $x_1, x_2 : ((\gamma, \delta) : y(\Gamma.\Delta)) \rightarrow X(\gamma, \delta)$ of parallel elements of X , then there exists a most general unifier $\rho : \Omega \rightarrow \Gamma$, such that either $\rho = \text{id}$ or the length of Ω is less than the length of Γ .* ◀

Proof. See Appendix A. ◀

► **Remark 31.** Note that Theorem 30 implies that the families

$$\begin{aligned} (\Delta : \text{Ty}_{\mathcal{C}}^*) \times (f, g : \text{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow y(\Xi)) & \mapsto f = g \\ (\Delta : \text{Ty}_{\mathcal{C}}^*) \times (A, B : \text{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \text{Ty}_{\mathcal{C}}) & \mapsto A = B \\ (\Delta : \text{Ty}_{\mathcal{C}}^*) \times (A : \text{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \text{Ty}_{\mathcal{C}}) \times (a, b : \forall \delta \rightarrow \text{Tm}_{\mathcal{C}}(A(\delta))) & \mapsto a = b \end{aligned}$$

are locally familially representable. Indeed, their categories of elements are the categories of unifiers for substitutions, types or terms. By Theorem 30, these categories are either empty, or admit a terminal object. In particular, every connected component admits a terminal object. ◻

4.2 Most general generalizations

We now apply first-order unification to the construction of most general generalizations.

We first describe this construction informally. For any type B over a unification context $\Gamma.\Delta$, we compute some B_0 over a context of the form $\Gamma_0.\Delta_0$ and a substitution $f : \Gamma \rightarrow \Gamma_0$ such that $\Delta = \Delta_0[f]$ and $B = B_0[f^+]$. The type B_0 should be the *most general generalization* of B that retains the dependency on Δ .

The type B_0 is essentially obtained by removing the dependencies on Γ , that is by replacing the subterms of B that only depend on Γ by new variables; these new variables are collected in the new context Γ_0 . Because of the dependencies of the generating terms, it is not always possible to fully remove a subterm. We have to rely on first-order unification to determine which parts can be removed; some of the new variables may need to be instantiated to more precise terms.

We give examples involving the following generating types and terms.

$$\begin{aligned} \mathbf{X} & : \text{Ty} \\ \mathbf{Y} & : \text{Tm}(\mathbf{X}) \rightarrow \text{Ty} \\ \mathbf{f}_1 & : \text{Tm}(\mathbf{X}) \rightarrow \text{Tm}(\mathbf{X}) \\ \mathbf{f}_2 & : \text{Tm}(\mathbf{X}) \rightarrow \text{Tm}(\mathbf{X}) \rightarrow \text{Tm}(\mathbf{X}) \\ \mathbf{g} & : \forall (x : \text{Tm}(\mathbf{X})) (y : \text{Tm}(\mathbf{Y}(x))) \rightarrow \text{Tm}(\mathbf{X}) \\ \mathbf{h} & : \forall (x : \text{Tm}(\mathbf{X})) \rightarrow \text{Tm}(\mathbf{Y}(x)) \rightarrow \text{Tm}(\mathbf{Y}(\mathbf{f}_1(x))) \end{aligned}$$

We write x, y, z, \dots for the variables from Γ and $\bar{x}, \bar{y}, \bar{z}, \dots$ for the variables from Δ .

- Consider $B = \mathbf{Y}(\mathbf{f}_1(x))$ over $(x : \mathbf{X})$.
Then we can pose $B_0 = \mathbf{Y}(y)$ over $(y : \mathbf{X})$, we have $B = B_0[y \mapsto \mathbf{f}_1(x)]$.
- Consider $B = \mathbf{Y}(\mathbf{f}_1(\bar{x}))$ over $(\bar{x} : \mathbf{X})$.
Then we have to keep $B_0 = \mathbf{Y}(\mathbf{f}_1(\bar{x}))$.
- Consider $B = \mathbf{Y}(\mathbf{f}_2(\mathbf{f}_1(x), \mathbf{f}_1(\bar{y})))$ over $(x : \mathbf{X}, \bar{y} : \mathbf{X})$.
Then $B_0 = \mathbf{Y}(\mathbf{f}_2(z, \mathbf{f}_1(\bar{y})))$ over $(z : \mathbf{X}, \bar{y} : \mathbf{X})$; we have $B = B_0[z \mapsto \mathbf{f}_1(x)]$.

- Consider $B = \mathbf{Y}(\mathbf{g}(\mathbf{f}_1(x), \bar{y}))$ over $(x : \mathbf{X}, \bar{y} : \mathbf{Y}(\mathbf{f}_1(x)))$.
Then $B_0 = \mathbf{Y}(\mathbf{g}(z, \bar{y}))$ over $(z : \mathbf{X}, \bar{y} : \mathbf{Y}(z))$; we have $B = B_0[z \mapsto \mathbf{f}_1(x), \bar{y} \mapsto \bar{y}]$.
- Consider $B = \mathbf{Y}(\mathbf{g}(\mathbf{f}_1(x), \mathbf{h}(x, \bar{y})))$ over $(x : \mathbf{X}, \bar{y} : \mathbf{Y}(x))$.
The $B_0 = B$. We cannot prune the subterm $\mathbf{f}_1(x)$, because of the typing constraints of \mathbf{g} and \mathbf{h} .

► **Proposition 32.** *The families*

$$(\Delta : \mathbf{Ty}_{\mathcal{C}}^*) \times (\Xi : \mathbf{Ob}_{\mathcal{C}}) \quad \mapsto \quad (\mathbf{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \mathbf{Tm}_{\mathcal{C}}^*(\Xi)) \quad (1)$$

$$(\Delta : \mathbf{Ty}_{\mathcal{C}}^*) \quad \mapsto \quad (\mathbf{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \mathbf{Ty}_{\mathcal{C}}) \quad (2)$$

$$(\Delta : \mathbf{Ty}_{\mathcal{C}}^*) \times (A : \mathbf{Tm}_{\mathcal{C}}^*(\Delta) \rightarrow \mathbf{Ty}_{\mathcal{C}}) \mapsto \forall(\delta : \mathbf{Tm}_{\mathcal{C}}^*(\Delta)) \rightarrow \mathbf{Tm}_{\mathcal{C}}(A(\delta)) \quad (3)$$

are locally familially representable.

In particular the family $\mathbf{MonoSort}_{\mathcal{C}}$, which is the coproduct of the families (2) and (3), is locally familially representable.

Proof. See Appendix A. ◀

Proof of Theorem 2. This follows from Proposition 32 and Proposition 20. ◀

4.3 Strictification

► **Theorem 3.** *Let \mathcal{C} be a \mathbf{CwF} with weakly stable weak identity types. Then there exists a \mathbf{CwF} \mathcal{D} with stable weak identity types and a trivial fibration $F : \mathcal{D} \rightarrow \mathcal{C}$ in \mathbf{CwF} that weakly preserves identity types.*

Proof. Let \mathcal{D} be an I -cellular replacement of \mathcal{C} . We have a trivial fibration $F : \mathcal{D} \rightarrow \mathcal{C}$ in \mathbf{CwF} . By Proposition 14, \mathcal{D} can be equipped with weakly stable identity types \mathbf{ld} that are strictly preserved by F .

By Theorem 2, \mathcal{D} has familially representable polynomials sorts. Thus by Theorem 1, \mathcal{D} has stable identity types \mathbf{ld}^s that are weakly equivalent to the weakly stable identity types. In other words, the \mathbf{CwF} morphism $\mathbf{id} : (\mathcal{D}, \mathbf{ld}^s) \rightarrow (\mathcal{D}, \mathbf{ld})$ weakly preserves identity types. Then the composition $(\mathcal{D}, \mathbf{ld}^s) \xrightarrow{\mathbf{id}} (\mathcal{D}, \mathbf{ld}) \xrightarrow{F} (\mathcal{C}, \mathbf{ld})$ weakly preserves identity types. ◀

5 Other type-theoretic structures

So far we have only considered (weak) identity types. However our methods can more generally be applied to any weakly stable weak type-theoretic structure. Indeed the proofs of Theorem 1 and Theorem 3 only rely on Proposition 10 and on the fact that the parameters of the identity introduction and elimination structures can be specified by (closed) polynomial sorts. Thus the same proof scheme works for any type-theoretic structure that is weakly stable (in the sense that it satisfies a variant of Proposition 10). This holds in particular for most standard type-theoretic structures, including Π -types, Σ -types, coproducts, natural numbers and other inductive types, etc.

Note that in general, weak structures can only be specified in presence of identity types; thus their strictification depends on the strictification of identity types. It is then necessary to see Theorem 1 as a construction.

6

Towards full coherence theorems

We have presented general strictification methods for weakly stable weak type-theoretic structures. However we generally want coherence theorems that give a more precise comparison between the categories $\mathbf{CwF}_s^{\text{cxl}}$ and $\mathbf{CwF}_{ws}^{\text{cxl}}$ of contextual CwFs equipped with stable or weakly stable weak type-theoretic structures (for some unspecified choice of such structures).

Following [19, 16], we expect that these categories can be equipped with cofibrantly generated left-semi model structures, with trivial fibrations as defined in Definition 12. We then want to prove that the free-forgetful adjunction

$$\begin{array}{ccc}
 & L & \\
 & \curvearrowright & \\
 \mathbf{CwF}_{ws}^{\text{cxl}} & \perp & \mathbf{CwF}_s^{\text{cxl}} \\
 & \curvearrowleft & \\
 & R &
 \end{array}$$

is a Quillen equivalence. This notion of *Morita equivalence* between type theories has been studied by Isaev [17], albeit only for strictly stable type-theoretic structures.

We recall the definition of weak equivalence [19] between CwFs.

► **Definition 33.** *Let $F : \mathcal{C} \rightarrow \mathcal{D}$ be a CwF morphism, where \mathcal{D} is equipped with weakly stable weak identity types. The map F is a **weak equivalence** if it is essentially surjective on types and terms, i.e. if it satisfies the following weak type and term lifting conditions:*

(weak type lifting) *For every $\Gamma : \text{Ob}_{\mathcal{C}}$ and type $A : \mathfrak{y}(F(\Gamma)) \rightarrow \text{Ty}_{\mathcal{D}}$, there exists a type $A_0 : \mathfrak{y}(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}$ and an equivalence between $F(A_0)$ and A over $F(\Gamma)$.*

(weak term lifting) *For every $\Gamma : \text{Ob}_{\mathcal{C}}$, type $A : \mathfrak{y}(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}$ and term $a : (\gamma : \mathfrak{y}(F(\Gamma))) \rightarrow \text{Tm}_{\mathcal{D}}(F(A)(\gamma))$, there exists a term $a_0 : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \text{Tm}_{\mathcal{C}}(A(\gamma))$ and a typal equality between $F(a_0)$ and a over $F(\Gamma)$. ⌋*

► **Conjecture 34.** *The theories of weakly stable weak identity types and strictly stable weak identity types of are Morita equivalent: for every I_{ws} -cellular model $\mathcal{C} : \mathbf{CwF}_{ws}$, the unit $\eta : \mathcal{C} \rightarrow L(\mathcal{C})$ is a weak equivalence. ⌋*

Here the I_{ws} -cellular models are the freely generated models in \mathbf{CwF}_{ws} . Note that they do not coincide with the I -cellular CwFs.

We give an informal outline of a likely proof of this result. We leave a detailed proof to future work.

Fix a I_{ws} -cellular model $\mathcal{C} : \mathbf{CwF}_{ws}$. Since \mathcal{C} is freely generated, it admits a syntactic description and satisfies a universal property; a morphism $\mathcal{C} \rightarrow \mathcal{E}$ in \mathbf{CwF}_{ws} is determined by the image of the generating types and terms.

By Theorem 3, or a generalization to additional type formers, we have a CwF \mathcal{D} , equipped with strictly stable type structures, along with a trivial fibration $F : \mathcal{D} \rightarrow \mathcal{C}$ in \mathbf{CwF} that weakly preserves the various type structures.

By induction on the syntax of \mathcal{C} , we construct a morphism $G : \mathcal{C} \rightarrow \mathcal{D}$ in \mathbf{CwF}_{ws} along with a homotopy $\alpha : F \circ G \sim \text{id}_{\mathcal{C}}$. In other words, we construct a homotopy section G of F . If F was a morphism in \mathbf{CwF}_{ws} , we could obtain a (strict) section from the fact that \mathcal{C} is cofibrant in \mathbf{CwF}_{ws} and satisfies a strict lifting property with respect to trivial fibrations. Since F only preserves the type-theoretic structures weakly, we can only construct a homotopy section.

More precisely, this induction can be described using the *homotopical gluing* of $F : \mathcal{D} \rightarrow \mathcal{C}$; it is a model $\mathcal{G} : \mathbf{CwF}_{ws}$ that classifies the homotopy sections of F . Its objects are triples (Γ, Δ, e) , where $\Gamma : \text{Ob}_{\mathcal{D}}$, $\Delta : \text{Ob}_{\mathcal{C}}$ and e is an equivalence between $F(\Delta)$ and Γ . Its construction ought to be similar to other constructions of homotopical gluing models [25] and homotopical diagram models [20].

The universal property of \mathcal{C} then provides a section of $\pi_2 : \mathcal{G} \rightarrow \mathcal{C}$, which can be decomposed into a morphism $G : \mathcal{C} \rightarrow \mathcal{D}$ and a homotopy $\alpha : F \circ G \sim \text{id}_{\mathcal{C}}$.

$$\begin{array}{ccc} \mathcal{G} & \xrightarrow{\pi_1} & \mathcal{D} \\ \langle G, \text{id}, \alpha \rangle \uparrow \downarrow \pi_2 & & \\ \mathcal{C} & \xleftarrow{F} & \end{array}$$

By the universal property of $L(\mathcal{C})$, we obtain a map $T : L(\mathcal{C}) \rightarrow \mathcal{D}$ in \mathbf{CwF}_s such that $T \circ \eta = G$.

$$\begin{array}{ccc} \mathcal{D} & \xleftarrow{T} & \\ G \uparrow \downarrow F & & \\ \mathcal{C} & \xrightarrow{\eta} & L(\mathcal{C}) \end{array}$$

We can now attempt to prove the weak type lifting property for η . For any context $\Gamma : \text{Ob}_{\mathcal{C}}$ and type $A : y(\eta\Gamma) \rightarrow \text{Ty}_{L(\mathcal{C})}$ of $L(\mathcal{C})$, we have a candidate lift $F(T(A))[\alpha] : y(\Gamma) \rightarrow \text{Ty}_{\mathcal{C}}$. It remains to prove that $\eta(F(T(A))[\alpha])$ is equivalent to A , or equivalently that $\eta(F(T(A)))$ is equivalent to A over the context equivalence $\eta(\alpha_{\Gamma}) : \eta(F(G(\Gamma))) \cong \eta(\Gamma)$.

It suffices to construct a homotopy $\beta : \eta \circ F \circ T \sim \text{id}_{L(\mathcal{C})}$ along with a higher homotopy γ between the homotopies $\beta \circ \eta$ and $\eta \circ \alpha$. We expect that these homotopies can be constructed using the universal properties of respectively $L(\mathcal{C})$ and \mathcal{C} , by mapping into some other homotopical gluing models. The weak term lifting property also follows from the existence of these homotopies.

Thus, we have essentially reduced the proof of the Morita equivalence between weakly stable and strictly stable structures to the construction of three homotopical gluing models.

References

- 1 Steve Awodey and Michael A. Warren. Homotopy theoretic models of identity types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 146(1):45–55, 2009. doi:10.1017/S0305004108001783.
- 2 Martin E. Bidlingmaier. An interpretation of dependent type theory in a model category of locally cartesian closed categories. *CoRR*, abs/2007.02900, 2020. arXiv:2007.02900.
- 3 Rafaël Bocquet. Coherence of strict equalities in dependent type theories. *CoRR*, abs/2010.14166, 2020. arXiv:2010.14166.
- 4 Aurelio Carboni and Peter Johnstone. Connected limits, familial representability and artin glueing. *Mathematical Structures in Computer Science*, 5(4):441–459, 1995. doi:10.1017/S0960129500001183.
- 5 Aurelio Carboni and Peter Johnstone. Corrigenda for ‘connected limits, familial representability and artin glueing’. *MSCS. Mathematical Structures in Computer Science*, 14, February 2004. doi:10.1017/S0960129503004080.
- 6 John Cartmell. Generalised algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986. doi:10.1016/0168-0072(86)90053-9.
- 7 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Unityped, simply typed, and dependently typed. In *Joachim Lambek: The Interplay of Mathematics, Logic, and Linguistics*, pages 135–180. Springer, 2021.
- 8 Pierre-Louis Curien. Substitution up to Isomorphism. *Fundam. Informaticae*, 19(1/2):51–85, 1993.
- 9 Pierre-Louis Curien, Richard Garner, and Martin Hofmann. Revisiting the categorical interpretation of dependent type theory. *Theor. Comput. Sci.*, 546:99–119, 2014. doi:10.1016/j.tcs.2014.03.003.

- 10 Peter Dybjer. Internal type theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs, International Workshop TYPES'95, Torino, Italy, June 5-8, 1995, Selected Papers*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer, 1995. doi:10.1007/3-540-61780-9_66.
- 11 Nicola Gambino and Richard Garner. The identity type weak factorisation system. *Theor. Comput. Sci.*, 409(1):94–109, 2008. doi:10.1016/j.tcs.2008.08.030.
- 12 Nicola Gambino and Simon Henry. Towards a constructive simplicial model of Univalent Foundations. *Journal of the London Mathematical Society*, 105, 2022.
- 13 Joseph A. Goguen. What is unification? - a categorical view of substitution, equation and solution. In *Resolution of Equations in Algebraic Structures, Volume 1: Algebraic Techniques*, pages 217–261. Academic, 1989.
- 14 Martin Hofmann. On the interpretation of type theory in locally cartesian closed categories. In Leszek Pacholski and Jerzy Tiuryn, editors, *Computer Science Logic, 8th International Workshop, CSL '94, Kazimierz, Poland, September 25-30, 1994, Selected Papers*, volume 933 of *Lecture Notes in Computer Science*, pages 427–441. Springer, 1994. doi:10.1007/BFb0022273.
- 15 Martin Hofmann. *Syntax and semantics of dependent types*, pages 13–54. Springer London, London, 1997. doi:10.1007/978-1-4471-0963-1_2.
- 16 Valery Isaev. Model structures on categories of models of type theories. *Mathematical Structures in Computer Science*, 28:1695–1722, 2017.
- 17 Valery Isaev. Morita equivalences between algebraic dependent type theories. *CoRR*, abs/1804.05045, 2018. arXiv:1804.05045.
- 18 Valery Isaev. Indexed type theories. *Math. Struct. Comput. Sci.*, 31(1):3–63, 2021. doi:10.1017/S0960129520000092.
- 19 Chris Kapulkin and Peter Lumsdaine. The homotopy theory of type theories. *Advances in Mathematics*, 337, September 2016. doi:10.1016/j.aim.2018.08.003.
- 20 Krzysztof Kapulkin and Peter Lumsdaine. Homotopical inverse diagrams in categories with attributes. *Journal of Pure and Applied Algebra*, 225:106563, April 2021. doi:10.1016/j.jpaa.2020.106563.
- 21 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.
- 22 Peter LeFanu Lumsdaine and Michael A. Warren. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Trans. Comput. Log.*, 16(3):23:1–23:31, 2015. doi:10.1145/2754931.
- 23 Paige Randall North. Identity types and weak factorization systems in Cauchy complete categories. *Math. Struct. Comput. Sci.*, 29(9):1411–1427, 2019. doi:10.1017/S0960129519000033.
- 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 Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, 25(5):1203–1277, 2015. doi:10.1017/S0960129514000565.
- 26 Taichi Uemura. A general framework for the semantics of type theory. *CoRR*, abs/1904.04097, 2019. arXiv:1904.04097.
- 27 Taichi Uemura. *Abstract and concrete type theories*. PhD thesis, Institute for Logic, Language and Computation, 2021. URL: <https://dare.uva.nl/search?identifier=41ff0b60-64d4-4003-8182-c244a9afab3b>.
- 28 Benno van den Berg. Path categories and propositional identity types. *ACM Trans. Comput. Log.*, 19(2):15:1–15:32, 2018. doi:10.1145/3204492.

A Constructions and proofs

Details of Construction 16. Since \mathcal{C} is I -cellular, it is the colimit of a sequence

$$(\iota_i^{i+1} : \mathcal{C}_i \rightarrow \mathcal{C}_{i+1})_{i < \omega}$$

of basic I -cellular maps, with $\mathcal{C}_0 = \mathbf{0}_{\mathbf{CwF}}$ and $\mathcal{C}_\omega = \mathcal{C}$. When $i \leq j \leq \omega$, we write $\iota_i^j : \mathcal{C}_i \rightarrow \mathcal{C}_j$ for the composition of maps of that sequence.

For each $i \leq \omega$, the map $\iota_i^{i+1} : \mathcal{C}_i \rightarrow \mathcal{C}_{i+1}$ is a basic I -cellular map, specified by a set GenTy_i of generating types and a set GenTm_i of generating terms. For every $\mathcal{S} : \text{GenTy}_i$, we have a boundary $\partial\mathcal{S} : \text{Ob}_{\mathcal{C}_i}$ and a generating type $\mathcal{S} : \mathfrak{y}(\iota_i^{i+1}(\partial\mathcal{S})) \rightarrow \text{Ty}_{\mathcal{C}_{i+1}}$. For every $\mathbf{f} : \text{GenTm}_i$, we have a boundary $\partial\mathbf{f} : \text{Ob}_{\mathcal{C}_i}$, an output type $T\mathbf{f} : \mathfrak{y}(\partial\mathbf{f}) \rightarrow \text{Ty}_{\mathcal{C}_i}$ and a generating term $\mathbf{a} : (\gamma : \mathfrak{y}(\iota_i^{i+1}(\partial\mathbf{f}))) \rightarrow \text{Tm}_{\mathcal{C}_{i+1}}(\iota_i^{i+1}(T\mathbf{f})(\gamma))$. A morphism $F : \mathcal{C}_{i+1} \rightarrow \mathcal{E}$ is uniquely determined by the composition $F \circ \iota_i^{i+1}$ and by the image of the generating types and terms.

We pose $\text{GenTy}_{\mathcal{C}} \triangleq \coprod_{i < \omega} \text{GenTy}_i$ and $\text{GenTm}_{\mathcal{C}} \triangleq \coprod_{i < \omega} \text{GenTm}_i$. The boundaries and output types of $\text{GenTy}_{\mathcal{C}}$ and $\text{GenTm}_{\mathcal{C}}$ are defined in the evident way using the boundaries and output types of GenTy_i and GenTm_i . ◀

Details of Construction 17. This is a standard normalization proof, although it is easier than usual thanks to the absence of definitional equalities.

We first prove the existence of normal forms. We define a new CwF \mathcal{C}_{nf} ; its substitutions, types and terms are those of \mathcal{C} equipped with normal forms. We omit the full definition of \mathcal{C}_{nf} , it is lengthy but straightforward. It involves the definition of the action of normal substitutions on normal forms.

We have a projection morphism $F : \mathcal{C}_{\text{nf}} \rightarrow \mathcal{C}$. We then construct a section G of F , by transfinite induction on $i \leq \omega$. The precise induction hypothesis is that for any $i \leq \omega$, we construct a morphism $G_i : \mathcal{C}_i \rightarrow \mathcal{C}_{\text{nf}}$ such that $F \circ G_i = \iota_i^\omega$. The zero and limit cases are straightforward, and in the successor case we only have to show that the generating types and terms admit a normal form. This holds essentially by definition of normal forms. By definition of \mathcal{C}_{nf} , the section G equips every type A , term a or substitution σ with a normal form $\text{nf}(A)$, $\text{nf}(a)$ or $\text{nf}(\sigma)$.

In order to prove uniqueness, we prove that normalization is stable, i.e. that for every normal form $A^{\text{nf}} : \text{NfTy}(A)$, $a^{\text{nf}} : \text{Nf}(a)$ or $\sigma^{\text{nf}} : \text{Nf}^*(\sigma)$, we have $A^{\text{nf}} = \text{nf}(A)$, $a^{\text{nf}} = \text{nf}(a)$ or $\sigma^{\text{nf}} = \text{nf}(\sigma)$. This is shown by induction on normal forms. Most cases are straightforward. In the case of a generating type or term coming from the basic I -cellular map $\mathcal{C}_i \rightarrow \mathcal{C}_{i+1}$, we use the definition of G_{i+1} on these generating types and terms. ◀

Proof of Proposition 25. We prove by induction on closed polynomial sorts that for every $P : \text{PolySort}_{\mathcal{C}_1}$, the presheaf $\text{Elem}(P)$ is familiarly representable.

Case $P = \diamond$:

Then $\text{Elem}(P)$ is the terminal presheaf, which is represented by the terminal object of \mathcal{C} .

Case $P = Q.M$:

Here $M : \text{Elem}(Q) \rightarrow \text{MonoSort}_{\mathcal{C}_1}$ is a monomial sort over Q .

Take an element $\langle q, a \rangle : \mathfrak{y}(\Gamma) \rightarrow \text{Elem}(Q.M)$. Our goal is to construct the most general generalization of $\langle q, a \rangle$, i.e. a terminal object of the connected component of $\langle q, a \rangle$ in the category of elements of $\text{Elem}(Q.M)$.

By the induction hypothesis, we have a most general generalization $q_0 : \mathfrak{y}(\Gamma_0) \rightarrow \text{Elem}(Q)$ of q . By its universal property, there is a unique map $f : \Gamma \rightarrow \Gamma_0$ such that $q = q_0[f]$.

We now inspect $M[q_0] : \mathfrak{y}(\Gamma_0) \rightarrow \text{MonoSort}_{\mathcal{C}_1}$, noting that $a : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \text{Elem}(M[q_0](f(\gamma)))$.

Case $M[q_0] = \lambda\gamma \mapsto [\Delta(\gamma) \vdash \mathbb{t}y]$:

Here $\Delta : \mathfrak{y}(\Gamma_0) \rightarrow \mathfrak{Ty}_1^*$ is a telescope over Γ_0 .

We know that $a : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathfrak{Tm}_1^*(\Delta(f(\gamma))) \rightarrow \mathfrak{Ty}_1$. By definition of the presheaf \mathfrak{Ty}_1 , this means that we have a local universe (V, E) and a classifying map

$$\chi : (\gamma : \mathfrak{y}(\Gamma)) \rightarrow \mathfrak{Tm}_1^*(\Delta(f(\gamma))) \rightarrow \mathfrak{y}(V)$$

such that $a = \lambda(\gamma, \delta) \mapsto E(\chi(\gamma, \delta))$.

By condition (LF), there exists an object Γ_1 representing the presheaf

$$(\gamma : \mathfrak{y}(\Gamma_0)) \times (v : \mathfrak{Tm}_1^*(\Delta(f(\gamma))) \rightarrow \mathfrak{y}(V)).$$

We now define $\langle q_1, a_1 \rangle : \mathfrak{y}(\Gamma_1) \rightarrow \text{Elem}(Q.M)$:

$$\begin{aligned} q_1(\gamma, v) &\triangleq q_0(\gamma), \\ a_1(\gamma, v) &\triangleq \lambda(\delta : \mathfrak{Tm}_1^*(\Delta(f(\gamma)))) \mapsto E(v(\delta)). \end{aligned}$$

We have $\langle q, a \rangle = \langle q_1, a_1 \rangle[\langle f, \chi \rangle]$. By the universal properties of Γ_1 and q_0 , the element $\langle q_1, a_1 \rangle$ is the most general generalization of $\langle q, a \rangle$.

Case $M[q_0] = \lambda\gamma \mapsto [\delta : \Delta(\gamma) \vdash \mathbb{t}m(A(\gamma, \delta))]$:

Here $\Delta : \mathfrak{y}(\Gamma_0) \rightarrow \mathfrak{Ty}_1^*$ is a telescope over Γ_0 and $A : (\gamma : \mathfrak{y}(\Gamma_0)) \rightarrow \mathfrak{Tm}_1^*(\Delta(\gamma)) \rightarrow \mathfrak{Ty}_1$.

We can decompose A into a local universe (V, E) and a classifying map

$$\chi : (\gamma : \mathfrak{y}(\Gamma_0))(\delta : \mathfrak{Tm}_1^*(\Delta(\gamma))) \rightarrow \mathfrak{y}(V)$$

such that $A = \lambda(\gamma, \delta) \mapsto E(\chi(\gamma, \delta))$.

We know that $a : (\gamma : \mathfrak{y}(\Gamma))(\delta : \mathfrak{Tm}_1^*(\Delta[f])) \rightarrow \mathfrak{Tm}_C(E(\chi(f(\gamma), \delta)))$.

By condition (LF), there exists an object Γ_1 representing the presheaf

$$(\gamma : \mathfrak{y}(\Gamma_0)) \times (x : (\delta : \mathfrak{Tm}_1^*(\Delta(f(\gamma)))) \rightarrow \mathfrak{Tm}_C(E(\chi(\gamma, \delta)))).$$

We now define $\langle q_1, a_1 \rangle : \mathfrak{y}(\Gamma_1) \rightarrow \text{Elem}(Q.M)$:

$$\begin{aligned} q_1(\gamma, x) &\triangleq q_0(\gamma), \\ a_1(\gamma, x) &\triangleq \lambda(\delta : \mathfrak{Tm}_1^*(\Delta(f(\gamma)))) \mapsto x(\delta). \end{aligned}$$

We have $\langle q, a \rangle = \langle q_1, a_1 \rangle[\langle f, a \rangle]$. By the universal properties of Γ_1 and q_0 , the element $\langle q_1, a_1 \rangle$ is the most general generalization of $\langle q, a \rangle$. \blacktriangleleft

Proof of Theorem 30. By nested inductions first on the length of Γ , and then on the normal form of the substitution, type, or term x_1 .

Case $(\Gamma = \diamond)$: Let $\sigma : \Theta \rightarrow \diamond$ be a unifier of x_1 and x_2 . The map $\sigma = \langle \rangle$ is an epimorphism.

Thus $x_1 = x_2$ and $\text{id} : \Gamma \rightarrow \Gamma$ is the most general unifier of x_1 and x_2 .

Case $(X = \mathfrak{Tm}_C^*(\diamond))$:

In that case, $x_1 = x_2 = \langle \rangle$ and $\text{id} : \Gamma \rightarrow \Gamma$ is the most general unifier of x_1 and x_2 .

Case $(X = \mathfrak{Tm}_C^*(\Xi.A))$:

We can write $x_1 = \langle \xi_1, a_1 \rangle$ and $x_2 = \langle \xi_2, a_2 \rangle$. By the induction hypothesis for ξ_1 , we have a most general unifier $\rho : \Omega \rightarrow \Gamma$ of ξ_1 and ξ_2 .

If $\rho = \text{id}$, then a_1 and a_2 are parallel terms and by the induction hypothesis for a_1 we can find a most general unifier $\rho' : \Omega' \rightarrow \Gamma$ of a_1 and a_2 . It is then also a most general unifier of x_1 and x_2 .

Otherwise, the length of Ω is less than the length of Γ . By the induction hypothesis for Ω , we can then find a most general unifier $\rho' : \Omega' \rightarrow \Omega$ of $a_1[\rho]$ and $a_2[\rho]$. The composite $(\rho \circ \rho') : \Omega' \rightarrow \Gamma$ is then a most general unifier of x_1 and x_2 .

Case ($X = \mathbf{T}y_C$):

We can write $x_1 = \mathbf{S}[\sigma_1]$ for some generating type \mathbf{S} and $\sigma_1 : \Gamma.\Delta \rightarrow \partial\mathbf{S}$. Since x_1 and x_2 are unifiable, we can also write $x_2 = \mathbf{S}[\sigma_2]$ for some $\sigma_2 : \Gamma.\Delta \rightarrow \partial\mathbf{S}$. By the induction hypothesis for σ_1 , we have a most general unifier of σ_1 and σ_2 . It is then also a most general unifier of x_1 and x_2 .

Case ($X = \mathbf{T}m_C(A(-))$):

We have several subcases depending on the parallel terms x_1 and x_2 .

Case ($x_1 = \mathbf{f}[\sigma_1]$) and ($x_2 = \mathbf{g}[\sigma_2]$):

Since x_1 and x_2 are unifiable, $\mathbf{f} = \mathbf{g}$. Here $\sigma_1 : \Gamma.\Delta \rightarrow \partial\mathbf{f}$ and $\sigma_2 : \Gamma.\Delta \rightarrow \partial\mathbf{f}$. By the induction hypothesis for σ_1 , we have a most general unifier of σ_1 and σ_2 . It is then also a most general unifier of x_1 and x_2 .

If either x_1 or x_2 is a variable from Γ :

Without loss of generality, assume that x_1 is a variable from Γ . By Lemma 29, the term x_2 can be strengthened to only depend on Γ . If $x_1 = x_2$ then $\text{id} : \Gamma \rightarrow \Gamma$ is the most general unifier of x_1 and x_2 . Otherwise $x_1 \neq x_2$ and the instantiation $\Gamma[x_1 := x_2]$ is the most general unifier of x_1 and x_2 , by Lemma 28. The length of $\Gamma[x_1 := x_2]$ is then less than the length of Γ .

Otherwise, both x_1 and x_2 are variables from Δ :

Since x_1 and x_2 are unifiable by a substitution that preserves the variables from Δ , they have to be equal. Then $\text{id} : \Gamma \rightarrow \Gamma$ is the most general unifier of x_1 and x_2 . ◀

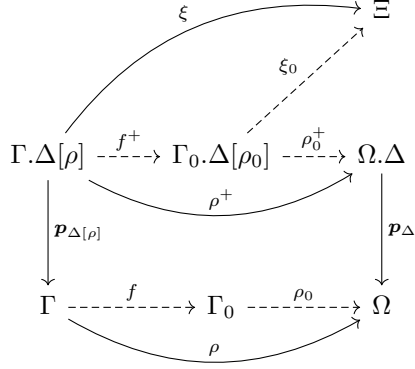
Proof of Proposition 32. The local familial representability can be unfolded to the following conditions:

Fix the following data:

- An object $\Gamma : \mathbf{Ob}_C$;
- An object Ω , a telescope $\Delta : y(\Omega) \rightarrow \mathbf{T}y_C^*$ and a map $\rho : \Gamma \rightarrow \Omega$;
- Either:
 - An object Ξ and a map $\xi : \Gamma.\Delta[\rho] \rightarrow \Xi$;
 - A type $A : y(\Gamma.\Delta[\rho]) \rightarrow \mathbf{T}y_C$;
 - A type $A : y(\Omega.\Delta) \rightarrow \mathbf{T}y_C$ and a term $a : (x : y(\Gamma.\Delta[\rho])) \rightarrow \mathbf{T}m_C(A(\rho^+(x)))$

Then we have to construct the following components, strictly naturally in Γ :

- An object $\Gamma_0 : \mathbf{Ob}_C$;
- A map $\rho_0 : \Gamma_0 \rightarrow \Omega$;
- Either:
 - A map $\xi_0 : \Gamma_0.\Delta[\rho_0] \rightarrow \Xi$;
 - A type $A_0 : y(\Gamma_0.\Delta[\rho_0]) \rightarrow \mathbf{T}y_C$;
 - A term $a_0 : (x : y(\Gamma_0.\Delta[\rho_0])) \rightarrow \mathbf{T}m_C(A(\rho_0^+(x)))$;
- Such that there exists a unique map $f : \Gamma \rightarrow \Gamma_0$ satisfying $\rho = \rho_0[f]$ and $\xi = \xi_0[f^+]$, $A = A_0[f^+]$ or $a = a_0[f^+]$.



We construct the most general generalizations by induction on the normal forms of ξ , A or a . The strict naturality in Γ will be proven in a second step.

Case ($\Xi = \diamond$) and ($\xi = \langle \rangle$):

We pose $\Gamma_0 = \Omega$, $\rho_0 = \text{id}$, $\xi_0 = \langle \rangle$ and $f = \rho$.

Case ($\Xi = \Theta.A$) and ($\xi = \langle \theta, a \rangle$):

In that case $\theta : \Gamma.\Delta[\rho] \rightarrow \Theta$ and $a : (x : y(\Gamma.\Delta[\rho])) \rightarrow A(\theta(x))$.

By the induction hypothesis for θ , we have $\Gamma_0, \rho_0 : \Gamma_0 \rightarrow \Omega$, $\theta_0 : \Gamma_0.\Delta[\rho_0] \rightarrow \Theta$ and there exists a unique map $f : \Gamma \rightarrow \Gamma_0$ such that $\rho_0[f] = \rho$ and $\theta_0[f^+] = \theta$.

By the induction hypothesis for a , we have $\Gamma_1, \rho_1 : \Gamma_1 \rightarrow \Gamma_0$, $a_1 : \Gamma_1.\Delta[\rho_0][\rho_1]$ and there is a unique map $g : \Gamma \rightarrow \Gamma_1$ such that $\rho_1[g] = f$ and $a_1[g^+] = a$.

We then pose $\Gamma_2 = \Gamma_1$, $\rho_2 = \rho_0 \circ \rho_1$ and $\xi_2 = \langle \theta_0[\rho_1], a_1 \rangle$. The map $g : \Gamma \rightarrow \Gamma_1$ is then the unique map such that $\rho_2[g] = \rho$ and $\xi_2[g^+] = \xi$.

Case ($A = \mathbf{S}[\sigma]$):

Here $\sigma : \Gamma \rightarrow \partial \mathbf{S}$. We just use the induction hypothesis for σ , and pose $A_0 = \mathbf{S}[\sigma_0]$.

Case ($a = a'[\mathbf{p}_{\Delta[\rho]}]$)

As a special case, we check if the term a depends on any variable from $\Delta[\rho]$. If it can be strengthened to a term a' over Γ such that $a'[\mathbf{p}_{\Delta[\rho]}] = a$, we also know that the type A cannot depend on any variable from Δ , and can be strengthened to $A' : y(\Omega) \rightarrow \text{Ty}_{\mathcal{C}}$ such that $A'[\mathbf{p}_{\Delta}] = A$. We then pose $\Gamma_0 = (\omega : \Omega).(a_0 : A'(\delta))$, $\rho_0 = \mathbf{p}_{A'} : \Gamma_0 \rightarrow \Omega$ and $f = \langle \rho, a' \rangle$.

Case ($\mathbf{Var}_{\Gamma.\Delta[\rho]}(a)$):

If a is a variable from $\Gamma.\Delta[\rho]$, then a has to be variable from $\Delta[\rho]$, as variables from Γ are dealt with in the case above.

Then we let a_0 be the corresponding variable from Δ and we pose $\Gamma_0 = \Omega$, $\rho_0 = \text{id}$ and $f = \rho$.

Case ($a = \mathbf{f}[\tau]$):

Here $\tau : \Gamma.\Delta[\rho] \rightarrow \partial \mathbf{f}$ and $a : (x : y(\Gamma.\Delta[\rho])) \rightarrow \text{Tm}_{\mathcal{C}}(\mathbf{Tf}(\tau(x)))$. We then know that $A[\rho^+] = \mathbf{Tf}[\tau]$.

By the induction hypothesis for τ , we have $\Gamma_0, \rho_0 : \Gamma_0 \rightarrow \Omega$, $\tau_0 : \Gamma_0.\Delta[\rho_0] \rightarrow \partial \mathbf{f}$ and there is a unique map $f : \Gamma \rightarrow \Gamma_0$ such that $\rho = \rho_0[f]$ and $\tau = \tau_0[f^+]$.

The types $A[\rho_0^+]$ and $\mathbf{Tf}[\tau_0]$ may differ. We know however that they are unifiable by the map f^+ ; thus by first-order unification (Theorem 30), we can find a most general unifier $\rho_1 : \Gamma_1 \rightarrow \Gamma_0$ of these two types. By the universal property of the most general unifier, we have a factorization of f as a map $g : \Gamma \rightarrow \Gamma_1$ followed by $\rho_1 : \Gamma_1 \rightarrow \Gamma_0$.

Now we pose $\Gamma_2 = \Gamma_1$, $\rho_2 = \rho_0 \circ \rho_1$, $a_2 = \mathbf{f}[\tau[\rho_1^+]]$. The map $g : \Gamma_1 \rightarrow \Gamma_0$ is then the unique map such that $\rho_2[g] = \rho$ and $a_2[g] = \tau$.

It remains to prove that the above construction is strictly natural in Γ : we have to prove for any ξ , A or a and any substitution $\sigma : \Lambda \rightarrow \Gamma$ that the most general generalizations of ξ and $\xi \circ \sigma$ (or A and $A[\sigma^+]$, or a and $a[\sigma^+]$) coincide. We prove this by induction on the normal forms of ξ , A or a , following the inductive cases of the previous construction. It is then straightforward to check that the construction follows the same cases for both ξ and $\xi[\sigma^+]$ (or A and $A[\sigma^+]$, or a and $a[\sigma^+]$).

The main subtlety happens when $a : \mathbf{y}(\Gamma.\Delta[\rho]) \rightarrow \mathbf{Tm}_C(-)$ is a variable from Γ . In that case, the substituted term $a[\sigma^+] : \mathbf{y}(\Lambda.\Delta[\rho][\sigma]) \rightarrow \mathbf{Tm}_C(-)$ is not necessarily a variable. However it can be strengthened to a term that only depends on Λ . Thus our construction of the most general generalization of both a and $a[\sigma^+]$ will use the special case for terms that don't depend on Δ . Without this special case, we would not be able to prove that our construction is strictly natural in Γ . ◀

A Machine-Checked Proof of Birkhoff’s Variety Theorem in Martin-Löf Type Theory

William DeMeo  

New Jersey Institute of Technology, Newark, NJ, USA

Jacques Carette  

McMaster University, Hamilton, Canada

Abstract

The Agda Universal Algebra Library is a project aimed at formalizing the foundations of universal algebra, equational logic and model theory in dependent type theory using Agda. In this paper we draw from many components of the library to present a self-contained, formal, constructive proof of Birkhoff’s HSP theorem in Martin-Löf dependent type theory. This achieves one of the project’s initial goals: to demonstrate the expressive power of inductive and dependent types for representing and reasoning about general algebraic and relational structures by using them to formalize a significant theorem in the field.

2012 ACM Subject Classification Theory of computation → Logic and verification; Computing methodologies → Representation of mathematical objects; Theory of computation → Type theory

Keywords and phrases Agda, constructive mathematics, dependent types, equational logic, formal verification, Martin-Löf type theory, model theory, universal algebra

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.4

Related Version *Full Version:* <https://arxiv.org/abs/2101.10166>

Supplementary Material *Software (Agda Sources):* <https://github.com/uilib/agda-algebras> archived at [swh:1:dir:29817e5c87bb55467269dad672f7f4b4152733d7](https://swh.1:dir:29817e5c87bb55467269dad672f7f4b4152733d7)

Funding *William DeMeo:* partially supported by ERC Consolidator Grant No. 771005.

Acknowledgements This work would not have been possible without the wonderful Agda language and the Agda Standard Library, developed and maintained by The Agda Team [21]. We thank the three anonymous referees for carefully reading the manuscript and offering many excellent suggestions which resulted in a vast improvement in the overall presentation. One referee went above and beyond and provided us with a simpler formalization of free algebras which led to simplifications of the proof of the main theorem. We are extremely grateful for this.

1 Introduction

The Agda Universal Algebra Library (agda-algebras) [8] formalizes the foundations of universal algebra in intensional Martin-Löf type theory (MLTT) using Agda [15, 18]. The library includes a collection of definitions and verified theorems originated in classical (set-theory based) universal algebra and equational logic, but adapted to MLTT.

The first major milestone of the project is a complete formalization of *Birkhoff’s variety theorem* (also known as the *HSP theorem*) [4]. To the best of our knowledge, this is the first time Birkhoff’s celebrated 1935 result has been formalized in MLTT.¹

Our first attempt to formalize Birkhoff’s theorem suffered from two flaws.² First, we assumed function extensionality in MLTT; consequently, it was unclear whether the formalization was fully constructive. Second, an inconsistency could be contrived by taking the

¹ An alternative formalization based on classical set-theory was achieved in [13].

² See the `Birkhoff.lagda` file in the `uilib/uilib.gitlab.io` repository (15 Jan 2021 commit 71f1738) [6].



type X , representing an arbitrary collection of variable symbols, to be the two element type (see §7.1 for details). To resolve these issues, we developed a new formalization of the HSP theorem based on *setoids* and rewrote much of the `agda-algebras` library to support this approach. This enabled us to avoid function extensionality altogether. Moreover, the type X of variable symbols was treated with more care using the *context* and *environment* types that Andreas Abel uses in [1] to formalize Birkhoff’s completeness theorem. These design choices are discussed further in §2.2–2.3.

What follows is a self-contained formal proof of the HSP theorem in `Agda`. This is achieved by extracting a subset of the `agda-algebras` library, including only the pieces needed for the proof, into a single literate `Agda` file.³ For spaces reasons, we elide some inessential parts, but strive to preserve the essential content and character of the development. Specifically, routine or overly technical components, as well as anything that does not seem to offer insight into the central ideas of the proof are omitted. (The file `src/Demos/HSP.lagda` mentioned above includes the full proof.)

In this paper, we highlight some of the more challenging aspects of formalizing universal algebra in type theory. To some extent, this is a sobering glimpse of the significant technical hurdles that must be overcome to do mathematics in dependent type theory. Nonetheless, we hope to demonstrate that MLTT is a relatively natural language for formalizing universal algebra. Indeed, we believe that researchers with sufficient patience and resolve can reap the substantial rewards of deeper insight and greater confidence in their results by using type theory and a proof assistant like `Agda`. On the other hand, this paper is probably not the best place to learn about the latter, since we assume the reader is already familiar with MLTT and `Agda`. In summary, our main contribution is to show that a straightforward but very general representation of algebraic structures in dependent type theory is quite practical, as we demonstrate by formalizing a major seminal result of universal algebra.

2 Preliminaries

2.1 Logical foundations

To best emulate MLTT, we use `{-# OPTIONS -without-K -exact-split -safe #-}`; *without-K* disables Streicher’s K axiom; *exact-split* directs `Agda` to accept only definitions behaving like *judgmental* equalities; *safe* ensures that nothing is postulated outright. (See [19, 20, 22].)

We also use some definitions from `Agda`’s standard library (ver. 1.7). As shown in Appendix §A, these are imported using the `open import` directive and they include some adjustments to “standard” `Agda` syntax. In particular, we use `Type` in place of `Set`, the infix long arrow symbol, `_→_`, in place of `Func` (the type of “setoid functions,” discussed in §2.3), and the symbol `_{\$}` in place of `f` (application of the map of a setoid function); we use `fst` and `snd`, and sometimes `|_` and `||_||`, to denote the first and second projections out of the product type `_×_`.

2.2 Setoids

A *setoid* is a pair consisting of a type and an equivalence relation on that type. Setoids are useful for representing a set with an explicit, “local” notion of equivalence, instead of relying on an implicit, “global” one as is more common in set theory. In reality, informal mathematical

³ `src/Demos/HSP.lagda` in the `agda-algebras` repository: github.com/ualib/agda-algebras

practice relies on equivalence relations quite pervasively, taking great care to define only functions that preserve equivalences, while eliding the details. To be properly formal, such details must be made explicit. While there are many different workable approaches, the one that requires no additional meta-theory is based on setoids, which is why we adopt it here. While in some settings setoids are found by others to be burdensome, we have not found them to be so for universal algebra.

The `agda-algebras` library was first developed without setoids, relying on propositional equality instead, along with some experimental, domain-specific types for equivalence classes, quotients, etc. This required postulating function extensionality,⁴ which is known to be independent from MLTT [9, 10]; this was unsatisfactory as we aimed to show that the theorems hold directly in MLTT without extra axioms. The present work makes no appeal to functional extensionality or classical axioms like `Choice` or `Excluded Middle`.

2.3 Setoid functions

A *setoid function* is a function from one setoid to another that respects the underlying equivalences. If \mathbf{A} and \mathbf{B} are setoids, we use $\mathbf{A} \rightarrow \mathbf{B}$ to denote the type of setoid functions from \mathbf{A} to \mathbf{B} . We define the *inverse* of such a function in terms of the image of the function's domain, as follows.

```

module _ {A : Setoid α ρa} {B : Setoid β ρb} where
  open Setoid B using ( _≈_ ; sym ) renaming ( Carrier to B )

  data Image_⇒_ (f : A → B) : B → Type (α ⊔ β ⊔ ρb) where
    eq : {b : B} → ∀ a → b ≈ f ($) a → Image f ⇒ b

```

An inhabitant of the `Image f ⇒ b` type is a point $a : \text{Carrier } \mathbf{A}$, along with a proof $p : b \approx f a$, that f maps a to b . Since a proof of `Image f ⇒ b` must include a concrete witness $a : \text{Carrier } \mathbf{A}$, we can actually *compute* a range-restricted right-inverse of f . Here is the definition of `Inv` accompanied by a proof that it gives a right-inverse.

```

Inv : (f : A → B) {b : B} → Image f ⇒ b → Carrier A
Inv _ (eq a _) = a

InvIsInverser : {f : A → B} {b : B} (q : Image f ⇒ b) → f ($) (Inv f q) ≈ b
InvIsInverser (eq _ p) = sym p

```

If $f : \mathbf{A} \rightarrow \mathbf{B}$ then we call f *injective* provided $\forall (a_0 a_1 : \mathbf{A}), f ($) a_0 \approx^B f ($) a_1$ implies $a_0 \approx^A a_1$; we call f *surjective* provided $\forall (b : \mathbf{B}) \exists (a : \mathbf{A})$ such that $f ($) a \approx^B b$. We omit the straightforward Agda definitions.

Factorization of setoid functions⁵

Any (setoid) function $f : \mathbf{A} \rightarrow \mathbf{B}$ factors as a surjective map `toIm` : $\mathbf{A} \rightarrow \text{Im } f$ followed by an injective map `fromIm` : $\text{Im } f \rightarrow \mathbf{B}$.

⁴ the axiom asserting that two point-wise equal functions are equal

⁵ The code in this paragraph was suggested by an anonymous referee.

4:4 A Machine-Checked Proof of Birkhoff's Theorem

```

module _ { A : Setoid  $\alpha$   $\rho^a$  } { B : Setoid  $\beta$   $\rho^b$  } where

  lm : (f : A  $\rightarrow$  B)  $\rightarrow$  Setoid _ _
  Carrier (lm f) = Carrier A
   $\approx^s$  _ (lm f) b1 b2 = f  $\langle$  $  $\rangle$  b1  $\approx$  f  $\langle$  $  $\rangle$  b2 where open Setoid B

  isEquivalence (lm f) = record { refl = refl ; sym = sym ; trans = trans }
    where open Setoid B

  tolm : (f : A  $\rightarrow$  B)  $\rightarrow$  A  $\rightarrow$  lm f
  tolm f = record { f = id ; cong = cong f }

  fromlm : (f : A  $\rightarrow$  B)  $\rightarrow$  lm f  $\rightarrow$  B
  fromlm f = record { f =  $\lambda$  x  $\rightarrow$  f  $\langle$  $  $\rangle$  x ; cong = id }

  fromlm-inj : (f : A  $\rightarrow$  B)  $\rightarrow$  IsInjective (fromlm f)
  fromlm-inj _ = id

  tolm-surj : (f : A  $\rightarrow$  B)  $\rightarrow$  IsSurjective (tolm f)
  tolm-surj _ = eq _ (refls B)

```

3 Basic Universal Algebra

We now develop a working vocabulary in MLTT corresponding to classical, single-sorted, set-based universal algebra. We cover a number of important concepts, but limit ourselves to those required to prove Birkhoff's HSP theorem. In each case, we give a type-theoretic version of the informal definition, followed by its Agda implementation.

This section is organized into the following subsections: §3.1 defines a general type of *signatures* of algebraic structures; §3.2 does the same for structures and their products; §3.3 defines *homomorphisms*, *monomorphisms*, and *epimorphisms*, presents types that codify these concepts, and formally verifies some of their basic properties; §3.5–3.6 do the same for *subalgebras* and *terms*, respectively.

3.1 Signatures

An (algebraic) *signature* is a pair $S = (F, \rho)$ where F is a collection of *operation symbols* and $\rho : F \rightarrow \mathbb{N}$ is an *arity function* which maps each operation symbol to its arity. Here, \mathbb{N} denotes the *arity type*. Heuristically, the arity ρf of an operation symbol $f \in F$ may be thought of as the number of arguments that f takes as “input.” We represent signatures as inhabitants of the following dependent pair type.

$$\text{Signature} : (\mathcal{V} : \text{Level}) \rightarrow \text{Type} (\text{Isuc} (\mathbb{N} \mathcal{V}))$$

$$\text{Signature} \mathcal{V} = \Sigma [F \in \text{Type} \mathbb{C}] (F \rightarrow \text{Type} \mathcal{V})$$

Recalling our syntax for the first and second projections, if S is a signature, then $| S |$ denotes the set of operation symbols and $\| S \|$ denotes the arity function. Thus, if $f : | S |$ is an operation symbol in the signature S , then $\| S \| f$ is the arity of f .

We need to augment our **Signature** type so that it supports algebras over setoid domains. To do so, following Abel [1], we define an operator that translates an ordinary signature into a *setoid signature*, that is, a signature over a setoid domain. This raises a minor technical issue:

given operations f and g , with arguments $u : \llbracket S \rrbracket f \rightarrow A$ and $v : \llbracket S \rrbracket g \rightarrow A$, respectively, and a proof of $f \equiv g$ (*intensional* equality), we ought to be able to check whether u and v are pointwise equal. Technically, u and v appear to inhabit different types; of course, this is reconciled by the hypothesis $f \equiv g$, as we see in the next definition (borrowed from [1]).

```
EqArgs : {S : Signature @ V}{ξ : Setoid α ρa}
  → ∀ {f g} → f ≡ g → (⊥ S ⊥ f → Carrier ξ) → (⊥ S ⊥ g → Carrier ξ) → Type (V ⊔ ρa)
EqArgs {ξ = ξ} ≡.refl u v = ∀ i → u i ≈ v i where open Setoid ξ using ( _≈_ )
```

This makes it possible to define an operator which translates a signature for algebras over bare types into a signature for algebras over setoids. We denote this operator by $\langle _ \rangle$.

```
⟨_⟩ : Signature @ V → Setoid α ρa → Setoid _ _
Carrier (⟨ S ⟩ ξ) = Σ[ f ∈ | S | ] (⊥ S ⊥ f → ξ .Carrier)
_≈s_ (⟨ S ⟩ ξ)(f , u)(g , v) = Σ[ eqv ∈ f ≡ g ] EqArgs{ξ = ξ} eqv u v
refle (isEquivalence (⟨ S ⟩ ξ)) = ≡.refl , λ i → refls ξ
syme (isEquivalence (⟨ S ⟩ ξ)) (≡.refl , g) = ≡.refl , λ i → syms ξ (g i)
transe (isEquivalence (⟨ S ⟩ ξ)) (≡.refl , g)(≡.refl , h) = ≡.refl , λ i → transs ξ (g i) (h i)
```

3.2 Algebras

An *algebraic structure* $\mathbf{A} = (A, F^A)$ in the signature $S = (F, \rho)$, or *S-algebra*, consists of

- a type A , called the *domain* of the algebra;
- a collection $F^A := \{ f^A \mid f \in F, f^A : (\rho f \rightarrow A) \rightarrow A \}$ of *operations* on A ;
- a (potentially empty) collection of *identities* satisfied by elements and operations of \mathbf{A} .

Our Agda implementation represents algebras as inhabitants of a record type with two fields – a **Domain** setoid denoting the domain of the algebra, and an **Interp** function denoting the interpretation in the algebra of each operation symbol in S . We postpone introducing identities until §4.

```
record Algebra α ρ : Type (⊙ ⊔ V ⊔ Isuc (α ⊔ ρ)) where
  field Domain : Setoid α ρ
  Interp : ⟨ S ⟩ Domain → Domain
```

Thus, for each operation symbol in S we have a setoid function f whose domain is a power of **Domain** and whose codomain is **Domain**. Further, we define some syntactic sugar to make our formalizations easier to read and reason about. Specifically, if \mathbf{A} is an algebra, then

- $\mathbb{D}[\mathbf{A}]$ denotes the **Domain** setoid of \mathbf{A} ,
- $\mathbb{U}[\mathbf{A}]$ is the underlying carrier of (the **Domain** setoid of) \mathbf{A} , and
- $f \hat{\ } \mathbf{A}$ denotes the interpretation of the operation symbol f in the algebra \mathbf{A} .

We omit the straightforward formal definitions (see [7] for details).

Universe levels of algebra types

Types belong to *universes*, which are structured in Agda as follows: $\text{Type } \ell : \text{Type } (\text{suc } \ell)$, $\text{Type } (\text{suc } \ell) : \text{Type } (\text{suc } (\text{suc } \ell))$, \dots ⁶ While this means that $\text{Type } \ell$ has type $\text{Type } (\text{suc } \ell)$, it does *not* imply that $\text{Type } \ell$ has type $\text{Type } (\text{suc } (\text{suc } \ell))$. In other words, Agda's

⁶ $\text{suc } \ell$ denotes the successor of ℓ in the universe hierarchy.

universes are *non-cumulative*. This can be advantageous as it becomes possible to treat size issues more generally and precisely. However, dealing with explicit universe levels can be daunting, and the standard literature (in which uniform smallness is typically assumed) offers little guidance. While in some settings, such as category theory, formalizing in **Agda** works smoothly with respect to universe levels (see [12]), in universal algebra the terrain is bumpier. Thus, it seems worthwhile to explain how we make use of universe lifting and lowering functions, available in the **Agda Standard Library**, to develop domain-specific tools for dealing with **Agda**'s non-cumulative universe hierarchy.

The **Lift** operation of the standard library embeds a type into a higher universe. Specializing **Lift** to our situation, we define a function **Lift-Alg** with the following interface.

```
Lift-Alg : Algebra  $\alpha$   $\rho^a$   $\rightarrow$  ( $\ell_0$   $\ell_1$  : Level)  $\rightarrow$  Algebra ( $\alpha \sqcup \ell_0$ ) ( $\rho^a \sqcup \ell_1$ )
```

Lift-Alg takes an algebra parametrized by levels **a** and ρ^a and constructs a new algebra whose carrier inhabits **Type** ($\alpha \sqcup \ell_0$) and whose equivalence inhabits **Rel Carrier** ($\rho^a \sqcup \ell_1$). To be useful, this lifting operation should result in an algebra with the same semantic properties as the one we started with. We will see in §3.4 that this is indeed the case.

Product Algebras

We define the *product* of a family of algebras as follows. Let ι be a universe and I : **Type** ι a type (the “indexing type”). Then $\mathcal{A} : I \rightarrow$ **Algebra** α ρ^a represents an *indexed family of algebras*. Denote by $\prod \mathcal{A}$ the *product of algebras* in \mathcal{A} (or *product algebra*), by which we mean the algebra whose domain is the Cartesian product $\prod i : I, \mathbb{D}[\mathcal{A} i]$ of the domains of the algebras in \mathcal{A} , and whose operations are those arising from pointwise interpretation in the obvious way: if **f** is a **J**-ary operation symbol and if $\mathbf{a} : \prod i : I, \mathbb{J} \rightarrow \mathbb{D}[\mathcal{A} i]$ is, for each $i : I$, a **J**-tuple of elements of the domain $\mathbb{D}[\mathcal{A} i]$, then we define the interpretation of **f** in $\prod \mathcal{A}$ by

$$(f \hat{\ } \prod \mathcal{A}) \mathbf{a} := \lambda (i : I) \rightarrow (f \hat{\ } \mathcal{A} i)(\mathbf{a} i).$$

Here is the formal definition of the product algebra type in **Agda**.

```
module _ { $\iota$  : Level}{ $I$  : Type  $\iota$ } where

prod : ( $\mathcal{A} : I \rightarrow$  Algebra  $\alpha$   $\rho^a$ )  $\rightarrow$  Algebra ( $\alpha \sqcup \iota$ ) ( $\rho^a \sqcup \iota$ )

Domain (prod  $\mathcal{A}$ ) = record { Carrier =  $\forall i \rightarrow \mathbb{U}[\mathcal{A} i]$ 
  ; _ $\approx$ _ =  $\lambda \mathbf{a} \mathbf{b} \rightarrow \forall i \rightarrow (\_ \approx^s \_ \mathbb{D}[\mathcal{A} i]) (\mathbf{a} i)(\mathbf{b} i)$ 
  ; isEquivalence =
    record { refl =  $\lambda i \rightarrow \text{refl}^e (\text{isEquivalence } \mathbb{D}[\mathcal{A} i])$ 
      ; sym =  $\lambda x i \rightarrow \text{sym}^e (\text{isEquivalence } \mathbb{D}[\mathcal{A} i])(x i)$ 
      ; trans =  $\lambda x y i \rightarrow \text{trans}^e (\text{isEquivalence } \mathbb{D}[\mathcal{A} i])(x i)(y i)$  }}

Interp (prod  $\mathcal{A}$ )  $\langle \$ \rangle$  (f ,  $\mathbf{a}$ ) =  $\lambda i \rightarrow (f \hat{\ } \mathcal{A} i) (\text{flip } \mathbf{a} i)$ 
cong (Interp (prod  $\mathcal{A}$ )) ( $\equiv$ .refl , f=g) =  $\lambda i \rightarrow \text{cong} (\text{Interp } (\mathcal{A} i)) (\equiv$ .refl , flip f=g i)
```

Evidently, the carrier of the product algebra type is indeed the (dependent) product of the carriers in the indexed family. The rest of the definitions are the “pointwise” versions of the underlying ones.

3.3 Structure preserving maps and isomorphism

Throughout the rest of the paper, unless stated otherwise, **A** and **B** will denote *S*-algebras inhabiting the types **Algebra** α ρ^a and **Algebra** β ρ^b , respectively.

A *homomorphism* (or “hom”) from \mathbf{A} to \mathbf{B} is a setoid function $h : \mathbb{D}[\mathbf{A}] \longrightarrow \mathbb{D}[\mathbf{B}]$ that is *compatible* with all basic operations; that is, for every operation symbol $f : |S|$ and all tuples $a : \parallel S \parallel f \rightarrow \mathbb{U}[\mathbf{A}]$, we have $h \langle \$ \rangle (f \hat{\ } \mathbf{A}) a \approx (f \hat{\ } \mathbf{B}) \lambda x \rightarrow h \langle \$ \rangle (a x)$.

It is convenient to first formalize “compatible” (`compatible-map-op`), representing the assertion that a given setoid function $h : \mathbb{D}[\mathbf{A}] \longrightarrow \mathbb{D}[\mathbf{B}]$ commutes with a given operation symbol f , and then generalize over operation symbols to yield the type (`compatible-map`) of compatible maps from (the domain of) \mathbf{A} to (the domain of) \mathbf{B} .

```
module _ (A : Algebra α ρa)(B : Algebra β ρb) where

compatible-map-op : (D[ A ] → D[ B ]) → | S | → Type _
compatible-map-op h f = ∀ {a} → h ⟨ $ ⟩ (f ^ A) a ≈ (f ^ B) λ x → h ⟨ $ ⟩ (a x)
  where open Setoid D[ B ] using ( _≈_ )

compatible-map : (D[ A ] → D[ B ]) → Type _
compatible-map h = ∀ {f} → compatible-map-op h f
```

Using these we define the property (`IsHom`) of being a homomorphism, and finally the type (`hom`) of homomorphisms from \mathbf{A} to \mathbf{B} .

```
record IsHom (h : D[ A ] → D[ B ]) : Type (⊙ ⊔ ℳ ⊔ α ⊔ ρa ⊔ ρb) where
  constructor mkhom
  field      compatible : compatible-map h

hom : Type _
hom = Σ (D[ A ] → D[ B ]) IsHom
```

Thus, an inhabitant of `hom` is a pair (h, p) consisting of a setoid function h , from the domain of \mathbf{A} to that of \mathbf{B} , along with a proof p that h is a homomorphism.

A *monomorphism* (resp. *epimorphism*) is an injective (resp. surjective) homomorphism. The `agda-algebras` library defines predicates `IsMon` and `IsEpi` for these, as well as `mon` and `epi` for the corresponding types.

```
record IsMon (h : D[ A ] → D[ B ]) : Type (⊙ ⊔ ℳ ⊔ α ⊔ ρa ⊔ ρb) where
  field isHom : IsHom h
        isInjective : IsInjective h
  HomReduct : hom
  HomReduct = h , isHom

mon : Type _
mon = Σ (D[ A ] → D[ B ]) IsMon
```

As with `hom`, the type `mon` is a dependent product type; each inhabitant is a pair consisting of a setoid function, say, h , along with a proof that h is a monomorphism.

```
record IsEpi (h : D[ A ] → D[ B ]) : Type (⊙ ⊔ ℳ ⊔ α ⊔ β ⊔ ρb) where
  field isHom : IsHom h
        isSurjective : IsSurjective h
  HomReduct : hom
  HomReduct = h , isHom

epi : Type _
epi = Σ (D[ A ] → D[ B ]) IsEpi
```

Composition of homomorphisms

The composition of homomorphisms is again a homomorphism, and similarly for epimorphisms and monomorphisms. The proofs of these facts are straightforward so we omit them, but give them the names `o-hom` and `o-epi` so we can refer to them below.

Two structures are *isomorphic* provided there are homomorphisms from each to the other that compose to the identity. We define the following record type to represent this concept.

```

module _ (A : Algebra α ρa) (B : Algebra β ρb) where
  open Setoid D[ A ] using () renaming ( _≈_ to _≈A_ )
  open Setoid D[ B ] using () renaming ( _≈_ to _≈B_ )

  record _≅_ : Type (C ⊔ V ⊔ α ⊔ β ⊔ ρa ⊔ ρb) where
    constructor mkiso
    field
      to      : hom A B
      from    : hom B A
      to~from : ∀ b → | to | ⟨$⟩ (| from | ⟨$⟩ b) ≈B b
      from~to : ∀ a → | from | ⟨$⟩ (| to | ⟨$⟩ a) ≈A a

```

The `agda-algebras` library also includes formal proof that the `to` and `from` maps are bijections and that `_≅_` is an equivalence relation, but we suppress these details.

Homomorphic images

We have found that a useful way to encode the concept of *homomorphic image* is to produce a witness, that is, a surjective hom. Thus we define the type of surjective homs and also record the fact that an algebra is its own homomorphic image via the identity hom.⁷

```

_IsHomImageOf_ : (B : Algebra β ρb)(A : Algebra α ρa) → Type _
B IsHomImageOf A = Σ[ φ ∈ hom A B ] IsSurjective | φ |

IdHomImage : {A : Algebra α ρa} → A IsHomImageOf A
IdHomImage {α = α}{A = A} = id , λ {y} → Image_∃_.eq y refl
  where open Setoid D[ A ] using ( refl )

```

Factorization of homomorphisms

Another theorem in the `agda-algebras` library, called `HomFactor`, formalizes the following factorization result: if $g : \text{hom } A \ B$, $h : \text{hom } A \ C$, h is surjective, and $\ker h \subseteq \ker g$, then there exists $\varphi : \text{hom } C \ B$ such that $g = \varphi \circ h$. A special case of this result that we use below is the fact that the setoid function factorization we saw above lifts to factorization of homomorphisms. Moreover, we associate a homomorphism h with its image – which is (the domain of) a subalgebra of the codomain of h – using the function `HomIm` defined below.⁸

```

module _ {A : Algebra α ρa}{B : Algebra β ρb} where

  HomIm : (h : hom A B) → Algebra _ _
  Domain (HomIm h) = Im | h |
  Interp (HomIm h) ⟨$⟩ (f , la) = (f ^ A) la
  cong (Interp (HomIm h)) {x1 , x2} {x1 , y2} (≡.refl , e) =

```

⁷ Here and elsewhere we use the shorthand `ov α := C ⊔ V ⊔ α`, for any level α .

⁸ The definition of `HomIm` was provided by an anonymous referee.


```

begin
  | h | ($) (Interp A ($) (x1 , x2)) ≈⟨ h-compatible ⟩
  Interp B ($) (x1 , λ x → | h | ($) x2 x) ≈⟨ cong (Interp B) (≡.refl , e) ⟩
  Interp B ($) (x1 , λ x → | h | ($) y2 x) ≈⟨ h-compatible ⟩
  | h | ($) (Interp A ($) (x1 , y2)) ■
  where open Setoid D[ B ] ; open SetoidReasoning D[ B ]
        open IsHom || h || renaming (compatible to h-compatible)

  toHomImlm : (h : hom A B) → hom A (HomImlm h)
  toHomImlm h = toImlm | h | , mkhom (refls D[ B ])

  fromHomImlm : (h : hom A B) → hom (HomImlm h) B
  fromHomImlm h = fromImlm | h | , mkhom (IsHom.compatible || h ||)

```

3.4 Lift-Alg is an algebraic invariant

The `Lift-Alg` operation neatly resolves the technical problem of universe non-cumulativity because isomorphism classes of algebras are closed under `Lift-Alg`.

```

module _ {A : Algebra α ρa}{ℓ : Level} where
  Lift-≅l : A ≅ (Lift-Algl A ℓ)
  Lift-≅l = mkiso ToLiftl FromLiftl (ToFromLiftl{A = A}) (FromToLiftl{A = A}{ℓ})
  Lift-≅r : A ≅ (Lift-Algr A ℓ)
  Lift-≅r = mkiso ToLiftr FromLiftr (ToFromLiftr{A = A}) (FromToLiftr{A = A}{ℓ})

  Lift-≅ : {A : Algebra α ρa}{ℓ ρ : Level} → A ≅ (Lift-Alg A ℓ ρ)
  Lift-≅ = ≅-trans Lift-≅l Lift-≅r

```

3.5 Subalgebras

We say that \mathbf{A} is a *subalgebra* of \mathbf{B} and write $\mathbf{A} \leq \mathbf{B}$ just in case \mathbf{A} can be *homomorphically embedded* in \mathbf{B} ; in other terms, $\mathbf{A} \leq \mathbf{B}$ iff there exists an injective hom from \mathbf{A} to \mathbf{B} .

```

≤ : Algebra α ρa → Algebra β ρb → Type _
A ≤ B = Σ[ h ∈ hom A B ] IsInjective | h |

```

The subalgebra relation is reflexive, by the identity monomorphism (and transitive by composition of monomorphisms, hence, a *preorder*, though we won't need this fact here).

```

≤-reflexive : {A : Algebra α ρa} → A ≤ A
≤-reflexive = id , id

```

We conclude this subsection with a simple utility function that converts a monomorphism into a proof of a subalgebra relationship.

```

mon→≤ : {A : Algebra α ρa}{B : Algebra β ρb} → mon A B → A ≤ B
mon→≤ {A = A}{B} x = mon→intohom A B x

```

3.6 Terms

Fix a signature S and let X denote an arbitrary nonempty collection of variable symbols. Such a collection is called a *context*. Assume the symbols in X are distinct from the operation symbols of S , that is $X \cap |S| = \emptyset$. A *word* in the language of S is a finite sequence of members of $X \cup |S|$. We denote the concatenation of such sequences by simple juxtaposition.

4:10 A Machine-Checked Proof of Birkhoff's Theorem

Let S_0 denote the set of nullary operation symbols of S . We define by induction on n the sets T_n of *words* over $X \cup |S|$ as follows: $T_0 := X \cup S_0$ and $T_{n+1} := T_n \cup \mathcal{T}_n$, where \mathcal{T}_n is the collection of all $f \ t$ such that $f : |S|$ and $t : \parallel S \parallel f \rightarrow T_n$. An S -*term* is a term in the language of S and the collection of all S -*terms* in the context X is $\text{Term } X := \bigcup_n T_n$.

In type theory, this translates to two cases: variable injection and applying an operation symbol to a tuple of terms. This represents each term as a tree with an operation symbol at each **node** and a variable symbol at each leaf **g**; hence the constructor names (**g** for “generator” and **node** for “node”) in the following inductively defined type.

```
data Term (X : Type χ) : Type (ov χ) where
  g : X → Term X
  node : (f : |S|)(t :  $\parallel S \parallel f \rightarrow \text{Term } X$ ) → Term X
```

The term algebra

We enrich the **Term** type to a setoid of S -terms, which will ultimately be the domain of an algebra, called the *term algebra in the signature S* . This requires an equivalence on terms.

```
module _ {X : Type χ} where
  data _≈_ : Term X → Term X → Type (ov χ) where
    rfl : {x y : X} → x ≡ y → (g x) ≈ (g y)
    gnl : ∀ {f}{s t :  $\parallel S \parallel f \rightarrow \text{Term } X$ } → (∀ i → (s i) ≈ (t i)) → (node f s) ≈ (node f t)
```

Below we denote by **≈-isEquiv** the easy (omitted) proof that **≈** is an equivalence relation.

For a given signature S and context X , we define the algebra **T** X , known as the *term algebra in S over X* . The domain of **T** X is **Term** X and, for each operation symbol $f : |S|$, we define $f \hat{\ } \mathbf{T} X$ to be the operation which maps each tuple $t : \parallel S \parallel f \rightarrow \text{Term } X$ of terms to the formal term $f \ t$.

```
TermSetoid : (X : Type χ) → Setoid _ _
TermSetoid X = record { Carrier = Term X ; _≈_ = _≈_ ; isEquivalence = ≈-isEquiv }

T : (X : Type χ) → Algebra (ov χ) (ov χ)
Algebra.Domain (T X) = TermSetoid X
Algebra.Interp (T X) ($) (f , ts) = node f ts
cong (Algebra.Interp (T X)) (≡.refl , ss≈ts) = gnl ss≈ts
```

Environments and interpretation of terms

Fix a signature S and a context X . An *environment* for **A** and X is a setoid whose carrier is a mapping from the variable symbols X to the domain $\mathbb{U}[\mathbf{A}]$ and whose equivalence relation is pointwise equality. Our formalization of this concept is the same as that of [1], which Abel uses to formalize Birkhoff's completeness theorem.

```
module Environment (A : Algebra α ℓ) where
  open Setoid  $\mathbb{U}[\mathbf{A}]$  using ( _≈_ ; refl ; sym ; trans )

  Env : Type χ → Setoid _ _
  Env X = record { Carrier = X →  $\mathbb{U}[\mathbf{A}]$ 
    ; _≈_ = λ ρ τ → (x : X) → ρ x ≈ τ x
    ; isEquivalence = record { refl = λ _ → refl
      ; sym = λ h x → sym (h x)
      ; trans = λ g h x → trans (g x)(h x) }}
```

The *interpretation* of a term *evaluated* in a particular environment is defined as follows.

```

[ ] : {X : Type X} (t : Term X) → (Env X) → D[ A ]
[ g x ] ($ ρ) = ρ x
[ node f args ] ($ ρ) = (Interp A) ($) (f, λ i → [ args i ] ($) ρ)
cong [ g x ] u ≈ v = u ≈ v x
cong [ node f args ] x ≈ y = cong (Interp A) (≡.refl, λ i → cong [ args i ] x ≈ y)

```

Two terms are proclaimed *equal* if they are equal for all environments.

```

Equal : {X : Type X} (s t : Term X) → Type _
Equal {X = X} s t = ∀ (ρ : Carrier (Env X)) → [ s ] ($) ρ ≈ [ t ] ($) ρ

```

Proof that `Equal` is an equivalence relation, and that the implication $s \simeq t \rightarrow \text{Equal } s \ t$ holds for all terms s and t , is also found in [1]. We denote the latter by $\simeq \rightarrow \text{Equal}$ in the sequel.

Compatibility of terms

We need to formalize two more concepts involving terms. The first (`comm-hom-term`) is the assertion that every term commutes with every homomorphism, and the second (`interp-prod`) is the interpretation of a term in a product algebra.

```

module _ {X : Type X} {A : Algebra α ρa} {B : Algebra β ρb} (hh : hom A B) where
  open Environment A using ( [ ] )
  open Environment B using () renaming ( [ ] to [ ]B )
  open Setoid D[ B ] using ( _ ≈ _ ; refl )
  private hfunc = | hh | ; h = _ ($)_ hfunc

  comm-hom-term : (t : Term X) (a : X → U[ A ]) → h ([ t ] ($) a) ≈ [ t ]B ($) (h o a)
  comm-hom-term (g x) a = refl
  comm-hom-term (node f t) a = begin
    h([ node f t ] ($) a) ≈ (compatible || hh ||)
    (f ^ B)(λ i → h([ t i ] ($) a)) ≈ (cong(Interp B)(≡.refl, λ i → comm-hom-term(t i) a))
    [ node f t ]B ($) (h o a) ■ where open SetoidReasoning D[ B ]

  module _ {X : Type X} {L : Level} {I : Type L} (sI : I → Algebra α ρa) where
    open Setoid D[ [ ] sI ] using ( _ ≈ _ )
    open Environment using ( [ ] ; ≈ → Equal )

    interp-prod : (p : Term X) → ∀ ρ → ([ [ ] sI ] p) ($) ρ ≈ λ i → ([ sI i ] p) ($) λ x → (ρ x) i
    interp-prod (g x) = λ ρ i → ≈ → Equal (sI i) (g x) (g x) ≈-isRefl λ _ → (ρ x) i
    interp-prod (node f t) = λ ρ → cong (Interp ([ ] sI)) (≡.refl, λ j k → interp-prod (t j) ρ k)

```

4 Equational Logic

4.1 Term identities, equational theories, and the \models relation

An *S-term equation* (or *S-term identity*) is an ordered pair (p, q) of *S*-terms, also denoted by $p \approx q$. We define an *equational theory* (or *algebraic theory*) to be a pair $T = (S, \mathcal{E})$ consisting of a signature S and a collection \mathcal{E} of *S*-term equations.⁹

⁹ Some authors reserve the term *theory* for a *deductively closed* set of equations, that is, a set of equations that is closed under entailment.

4:12 A Machine-Checked Proof of Birkhoff's Theorem

We say that the algebra \mathbf{A} *models* the identity $p \approx q$ and we write $\mathbf{A} \models p \approx q$ if for all $\rho : X \rightarrow \mathbb{D}[\mathbf{A}]$ we have $\llbracket p \rrbracket \langle \$ \rangle \rho \approx \llbracket q \rrbracket \langle \$ \rangle \rho$. In other words, when interpreted in the algebra \mathbf{A} , the terms p and q are equal no matter what values are assigned to variable symbols occurring in p and q . If \mathcal{K} is a class of algebras of a given signature, then we write $\mathcal{K} \models p \approx q$ and say that \mathcal{K} *models* the identity $p \approx q$ provided $\mathbf{A} \models p \approx q$ for every $\mathbf{A} \in \mathcal{K}$.

```

module _ {X : Type χ} where
  _|=~=_ : Algebra α ρa → Term X → Term X → Type _
  A |= p ≈ q = Equal p q where open Environment A

  _||=~=_ : Pred (Algebra α ρa) ℓ → Term X → Term X → Type _
  K |= p ≈ q = ∀ A → K A → A |= p ≈ q

```

We represent a set of term identities as a predicate over pairs of terms, and we denote by $\mathbf{A} \models \mathcal{E}$ the assertion that \mathbf{A} models $p \approx q$ for all $(p, q) \in \mathcal{E}$.

```

_||=~=_ : (A : Algebra α ρa) → Pred(Term X × Term X)(ov χ) → Type _
A |= ℰ = ∀ {p q} → (p, q) ∈ ℰ → Equal p q where open Environment A

```

An important property of the binary relation \models is *algebraic invariance* (i.e., invariance under isomorphism). We formalize this result as follows.

```

module _ {X : Type χ}{A : Algebra α ρa}{B : Algebra β ρb}(p q : Term X) where

|=l-invar : A |= p ≈ q → A ≅ B → B |= p ≈ q
|=l-invar Appq (mkiso fh gh f~g g~f) ρ = begin
  [ p ] ($) ρ ≈~< cong [ p ] (f~g ∘ ρ) >
  [ p ] ($) (f ∘ (g ∘ ρ)) ≈~< comm-hom-term fh p (g ∘ ρ) >
  f([ p ]A ($) (g ∘ ρ)) ≈~< cong | fh | (Appq (g ∘ ρ)) >
  f([ q ]A ($) (g ∘ ρ)) ≈~< comm-hom-term fh q (g ∘ ρ) >
  [ q ] ($) (f ∘ (g ∘ ρ)) ≈~< cong [ q ] (f~g ∘ ρ) >
  [ q ] ($) ρ ■
  where private f = _($)_ | fh | ; g = _($)_ | gh |
         open Environment A using () renaming ( [ ] to [ ]A )
         open Environment B using ( [ ] ) ; open SetoidReasoning D[ B ]

```

If \mathcal{K} is a class of S -algebras, the set of identities modeled by \mathcal{K} , denoted $\text{Th } \mathcal{K}$, is called the *equational theory* of \mathcal{K} . If \mathcal{E} is a set of S -term identities, the class of algebras modeling \mathcal{E} , denoted $\text{Mod } \mathcal{E}$, is called the *equational class axiomatized* by \mathcal{E} . We codify these notions in the next two definitions.

```

Th : {X : Type χ} → Pred (Algebra α ρa) ℓ → Pred(Term X × Term X) _
Th K = λ (p, q) → K |= p ≈ q

Mod : {X : Type χ} → Pred(Term X × Term X) ℓ → Pred (Algebra α ρa) _
Mod ℰ A = ∀ {p q} → (p, q) ∈ ℰ → Equal p q where open Environment A

```

4.2 The Closure Operators H, S, P and V

Fix a signature S , let \mathcal{K} be a class of S -algebras, and define

- $\mathbf{H } \mathcal{K} :=$ the class of all homomorphic images of members of \mathcal{K} ;
- $\mathbf{S } \mathcal{K} :=$ the class of all subalgebras of members of \mathcal{K} ;
- $\mathbf{P } \mathcal{K} :=$ the class of all products of members of \mathcal{K} .

H , S , and P are *closure operators* (expansive, monotone, and idempotent). A class \mathcal{K} of S -algebras is said to be *closed under the taking of homomorphic images* provided $H \mathcal{K} \subseteq \mathcal{K}$. Similarly, \mathcal{K} is *closed under the taking of subalgebras* (resp., *arbitrary products*) provided $S \mathcal{K} \subseteq \mathcal{K}$ (resp., $P \mathcal{K} \subseteq \mathcal{K}$). The operators H , S , and P can be composed with one another repeatedly, forming yet more closure operators. We represent these three closure operators in type theory as follows.

```

module _ {α ρa β ρb : Level} where
  private a = α ⊔ ρa

  H : ∀ ℓ → Pred(Algebra α ρa) (a ⊔ ov ℓ) → Pred(Algebra β ρb) _
  H _ ℓ B = Σ[ A ∈ Algebra α ρa ] A ∈ ℓ × B IsHomImageOf A

  S : ∀ ℓ → Pred(Algebra α ρa) (a ⊔ ov ℓ) → Pred(Algebra β ρb) _
  S _ ℓ B = Σ[ A ∈ Algebra α ρa ] A ∈ ℓ × B ≤ A

  P : ∀ ℓ ι → Pred(Algebra α ρa) (a ⊔ ov ℓ) → Pred(Algebra β ρb) _
  P _ ι ℓ B = Σ[ l ∈ Type ι ] (Σ[ s ∈ (l → Algebra α ρa) ] (∀ i → s i ∈ ℓ) × (B ≅ ∏ s))

```

Identities modeled by an algebra \mathbf{A} are also modeled by every homomorphic image of \mathbf{A} and by every subalgebra of \mathbf{A} . We refer to these facts as $\models\text{-H-invar}$ and $\models\text{-S-invar}$; their definitions are similar to that of $\models\text{-I-invar}$. An identity satisfied by all algebras in an indexed collection is also satisfied by the product of algebras in the collection. We refer to this fact as $\models\text{-P-invar}$.

A *variety* is a class of S -algebras that is closed under the taking of homomorphic images, subalgebras, and arbitrary products. If we define $V \mathcal{K} := H (S (P \mathcal{K}))$, then \mathcal{K} is a variety iff $V \mathcal{K} \subseteq \mathcal{K}$. The class $V \mathcal{K}$ is called the *varietal closure* of \mathcal{K} . Here is how we define V in type theory. (The explicit universe level declarations that appear in the definition are needed for disambiguation.)

```

module _ {α ρa β ρb γ ρc δ ρd : Level} where
  private a = α ⊔ ρa ; b = β ⊔ ρb

  V : ∀ ℓ ι → Pred(Algebra α ρa) (a ⊔ ov ℓ) → Pred(Algebra δ ρd) _
  V ℓ ι ℓ = H{γ}{ρc}{δ}{ρd} (a ⊔ b ⊔ ℓ ⊔ ι) (S{β}{ρb} (a ⊔ ℓ ⊔ ι) (P ℓ ι ℓ))

```

The classes $H \mathcal{K}$, $S \mathcal{K}$, $P \mathcal{K}$, and $V \mathcal{K}$ all satisfy the same term identities. We will only use a subset of the inclusions needed to prove this assertion.¹⁰ First, the closure operator H preserves the identities modeled by the given class; this follows almost immediately from the invariance lemma $\models\text{-H-invar}$.

```

module _ {X : Type χ}{ℓ : Pred(Algebra α ρa) (α ⊔ ρa ⊔ ov ℓ)}{p q : Term X} where
  H-id1 : ℓ ⊨ p ≈ q → H{β = α}{ρa}ℓ ℓ ⊨ p ≈ q
  H-id1 σ B (A , kA , BimgA) = ⊨-H-invar{p = p}{q} (σ A kA) BimgA

```

The analogous preservation result for S is a consequence of the invariance lemma $\models\text{-S-invar}$; the converse, which we call $S\text{-id2}$, has an equally straightforward proof.

```

S-id1 : ℓ ⊨ p ≈ q → S{β = α}{ρa}ℓ ℓ ⊨ p ≈ q
S-id1 σ B (A , kA , B≤A) = ⊨-S-invar{p = p}{q} (σ A kA) B≤A

```

¹⁰The others are included in the `Setoid.Varieties.Preservation` module of the `agda-algebras` library.

```
S-id2 : S ℓ ℒ ⊨ p ≈ q → ℒ ⊨ p ≈ q
S-id2 Spq A kA = Spq A (A , (kA , ≤-reflexive))
```

The `agda-algebras` library includes analogous pairs of implications for **P**, **H**, and **V**, called **P-id1**, **P-id2**, **H-id1**, etc. whose formalizations we suppress.

5 Free Algebras

5.1 The absolutely free algebra

The term algebra $\mathbf{T} X$ is the *absolutely free* S -algebra over X . That is, for every S -algebra \mathbf{A} , the following hold.

- Every function from X to $\mathbb{U}[\mathbf{A}]$ lifts to a homomorphism from $\mathbf{T} X$ to \mathbf{A} .
- That homomorphism is unique.

Here we formalize the first of these properties by defining the lifting function `free-lift` and its setoid analog `free-lift-func`, and then proving the latter is a homomorphism.¹¹

```
module _ {X : Type χ} {A : Algebra α ρa} (h : X → U[ A ]) where
  free-lift : U[ T X ] → U[ A ]
  free-lift (g x) = h x
  free-lift (node f t) = (f ^ A) λ i → free-lift (t i)

  free-lift-func : D[ T X ] → D[ A ]
  free-lift-func ($) x = free-lift x
  cong free-lift-func = flcong where
    open Setoid D[ A ] using ( _≈_ ) renaming ( reflexive to reflexiveA )
    flcong : ∀ {s t} → s ≈ t → free-lift s ≈ free-lift t
    flcong ( _≈_.rfl x ) = reflexiveA (≡.cong h x)
    flcong ( _≈_.gfl x ) = cong (Interp A) (≡.refl , λ i → flcong (x i))

  lift-hom : hom (T X) A
  lift-hom = free-lift-func ,
    mkhom λ {a} → cong (Interp A) (≡.refl , λ i → (cong free-lift-func){a i} ≈-isRefl)
```

It turns out that the interpretation of a term p in an environment η is the same as the free lift of η evaluated at p . We apply this fact a number of times in the sequel.

```
module _ {X : Type χ} {A : Algebra α ρa} where
  open Setoid D[ A ] using ( _≈_ ; refl )
  open Environment A using ( [ ] )

  free-lift-interp : (η : X → U[ A ])(p : Term X) → [ p ] ($) η ≈ (free-lift{A = A} η) p
  free-lift-interp η (g x) = refl
  free-lift-interp η (node f t) = cong (Interp A) (≡.refl , (free-lift-interp η) ∘ t)
```

5.2 The relatively free algebra

Given an arbitrary class \mathcal{K} of S -algebras, we cannot expect that $\mathbf{T} X$ belongs to \mathcal{K} . Indeed, there may be no free algebra in \mathcal{K} . Nonetheless, it is always possible to construct an algebra that is free for \mathcal{K} and belongs to the class $\mathbf{S}(\mathbf{P} \mathcal{K})$. Such an algebra is called a *relatively*

¹¹ For the proof of uniqueness, see the `Setoid.Terms.Properties` module of the `agda-algebras` library.

free algebra over X (relative to \mathcal{K}). There are several informal approaches to defining this algebra. We now describe the approach on which our formal construction is based and then we present the formalization.

Let $\mathbb{F}[X]$ denote the relatively free algebra over X . We represent $\mathbb{F}[X]$ as the quotient $\mathbf{T} X / \approx$ where $x \approx y$ if and only if $h x = h y$ for every homomorphism h from $\mathbf{T} X$ into a member of \mathcal{K} . More precisely, if $\mathbf{A} \in \mathcal{K}$ and $h : \mathbf{hom}(\mathbf{T} X) \mathbf{A}$, then h factors as $\mathbf{T} X \xrightarrow{h} \mathbf{HomIm} h \xrightarrow{\subseteq} \mathbf{A}$ and $\mathbf{T} X / \ker h \cong \mathbf{HomIm} h \leq \mathbf{A}$; that is, $\mathbf{T} X / \ker h$ is (isomorphic to) an algebra in $\mathcal{S} \mathcal{K}$. Letting $\approx := \bigcap \{ \theta \in \mathbf{Con} \mathbf{T} X \mid \mathbf{T} X / \theta \in \mathcal{S} \mathcal{K} \}$, observe that $\mathbb{F}[X] := \mathbf{T} X / \approx$ is a subdirect product of the algebras $\{ \mathbf{T} X / \ker h \}$ as h ranges over all homomorphisms from $\mathbf{T} X$ to algebras in \mathcal{K} . Thus, $\mathbb{F}[X] \in \mathbf{P}(\mathcal{S} \mathcal{K}) \subseteq \mathcal{S}(\mathbf{P} \mathcal{K})$. As we have seen, every map $\rho : X \rightarrow \mathbb{U}[\mathbf{A}]$ extends uniquely to a homomorphism $h : \mathbf{hom}(\mathbf{T} X) \mathbf{A}$ and h factors through the natural projection $\mathbf{T} X \rightarrow \mathbb{F}[X]$ (since $\approx \subseteq \ker h$) yielding a unique homomorphism from $\mathbb{F}[X]$ to \mathbf{A} extending ρ .

In Agda we construct $\mathbb{F}[X]$ as a homomorphic image of $\mathbf{T} X$ in the following way. First, given X we define \mathbf{C} as the product of pairs (\mathbf{A}, ρ) of algebras $\mathbf{A} \in \mathcal{K}$ along with environments $\rho : X \rightarrow \mathbb{U}[\mathbf{A}]$. To do so, we contrive an index type for the product; each index is a triple $(\mathbf{A}, \mathbf{p}, \rho)$ where \mathbf{A} is an algebra, \mathbf{p} is proof of $\mathbf{A} \in \mathcal{K}$, and $\rho : X \rightarrow \mathbb{U}[\mathbf{A}]$ is an arbitrary environment.

```

module FreeAlgebra (K : Pred (Algebra  $\alpha$   $\rho^a$ )  $\ell$ ) where
  private c =  $\alpha \sqcup \rho^a$  ;  $\iota = \mathbf{ov} \ c \sqcup \ell$ 
  J : { $\chi$  : Level}  $\rightarrow$  Type  $\chi \rightarrow$  Type ( $\iota \sqcup \chi$ )
  J X =  $\Sigma [ \mathbf{A} \in \mathbf{Algebra} \ \alpha \ \rho^a ] \ \mathbf{A} \in \mathcal{K} \times (X \rightarrow \mathbb{U}[\mathbf{A}])$ 

  C : { $\chi$  : Level}  $\rightarrow$  Type  $\chi \rightarrow$  Algebra ( $\iota \sqcup \chi$ )( $\iota \sqcup \chi$ )
  C X =  $\prod \{ l = J \ X \} \ \_ \sqcup \_$ 

```

We then define $\mathbb{F}[X]$ to be the image of a homomorphism from $\mathbf{T} X$ to \mathbf{C} as follows.

```

homC : (X : Type  $\chi$ )  $\rightarrow$  hom (T X) (C X)
homC X =  $\sqbracket$ -hom-co  $\_$  ( $\lambda$  i  $\rightarrow$  lift-hom (snd || i ||))

F[_] : { $\chi$  : Level}  $\rightarrow$  Type  $\chi \rightarrow$  Algebra (ov  $\chi$ )( $\iota \sqcup \chi$ )
F[X] = HomIm (homC X)

```

Observe that if the identity $\mathbf{p} \approx \mathbf{q}$ holds in all $\mathbf{A} \in \mathcal{K}$ (for all environments), then $\mathbf{p} \approx \mathbf{q}$ holds in $\mathbb{F}[X]$; equivalently, the pair (\mathbf{p}, \mathbf{q}) belongs to the kernel of the natural homomorphism from $\mathbf{T} X$ onto $\mathbb{F}[X]$. This natural epimorphism is defined as follows.

```

module FreeHom {K : Pred (Algebra  $\alpha$   $\rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \mathbf{ov} \ \ell$ )} where
  private c =  $\alpha \sqcup \rho^a$  ;  $\iota = \mathbf{ov} \ c \sqcup \ell$ 
  open FreeAlgebra K using ( F[_] ; homC )

  epiF[_] : (X : Type c)  $\rightarrow$  epi (T X) F[X]
  epiF[X] = | toHomIm (homC X) | , record { isHom = || toHomIm (homC X) ||
                                           ; isSurjective = toIm-surj | homC X | }

  homF[_] : (X : Type c)  $\rightarrow$  hom (T X) F[X]
  homF[X] = IsEpi.HomReduct || epiF[X] ||

```

Before formalizing the HSP theorem in the next section, we need to prove the following important property of the relatively free algebra: For every algebra \mathbf{A} , if $\mathbf{A} \models \mathbf{Th}(\mathcal{V} \mathcal{K})$, then there exists an epimorphism from $\mathbb{F}[\mathbf{A}]$ onto \mathbf{A} , where \mathbf{A} denotes the carrier of \mathbf{A} .

4:16 A Machine-Checked Proof of Birkhoff's Theorem

```

module _ { A : Algebra (α ⊔ ρa ⊔ ℓ)(α ⊔ ρa ⊔ ℓ) } { K : Pred(Algebra α ρa)(α ⊔ ρa ⊔ ov ℓ) } where
  private c = α ⊔ ρa ⊔ ℓ ; ι = ov c
  open FreeAlgebra K using ( F[_] ; C )
  open Setoid D[ A ] using ( refl ; sym ; trans ) renaming ( Carrier to A ; _≈_ to _≈A_ )

```

```

F-ModTh-epi : A ∈ Mod (Th K) → epi F[ A ] A
F-ModTh-epi A ∈ ModThK = φ , isEpi where

```

```

φ : D[ F[ A ] ] → D[ A ]
_($)_ φ = free-lift { A = A } id
cong φ { p } { q } pq = Goal
where
  lift-pq : (p , q) ∈ Th K
  lift-pq B × ρ = begin
    [ p ] ($ ) ρ ≈< free-lift-interp { A = B } ρ p >
    free-lift ρ p ≈< pq ( B , × , ρ ) >
    free-lift ρ q ≈< free-lift-interp { A = B } ρ q >
    [ q ] ($ ) ρ ■
    where open SetoidReasoning D[ B ] ; open Environment B using ( [ ] )

```

```

Goal : free-lift id p ≈A free-lift id q
Goal = begin
  free-lift id p ≈< free-lift-interp { A = A } id p >
  [ p ] ($ ) id ≈< A ∈ ModThK { p = p } { q } lift-pq id >
  [ q ] ($ ) id ≈< free-lift-interp { A = A } id q >
  free-lift id q ■
  where open SetoidReasoning D[ A ] ; open Environment A using ( [ ] )

```

```

isEpi : IsEpi F[ A ] A φ
isEpi = record { isHom = mkhom refl ; isSurjective = eq (g _) refl }

```

```

F-ModThV-epi : A ∈ Mod (Th (V ℓ ι K)) → epi F[ A ] A
F-ModThV-epi A ∈ ModThVK = F-ModTh-epi λ { p } { q } → Goal { p } { q }
where
  Goal : A ∈ Mod (Th K)
  Goal { p } { q } × ρ = A ∈ ModThVK { p } { q } (V-id1 ℓ { p = p } { q } ×) ρ

```

6 Birkhoff's Variety Theorem

Let \mathcal{K} be a class of algebras and recall that \mathcal{K} is a *variety* provided it is closed under homomorphisms, subalgebras and products; equivalently, $\mathbf{V} \mathcal{K} \subseteq \mathcal{K}$. (Observe that $\mathcal{K} \subseteq \mathbf{V} \mathcal{K}$ holds for all \mathcal{K} since \mathbf{V} is a closure operator.) We call \mathcal{K} an *equational class* if it is the class of all models of some set of identities.

Birkhoff's variety theorem, also known as the HSP theorem, asserts that \mathcal{K} is an equational class if and only if it is a variety. In this section, we present the statement and proof of this theorem – first in a style similar to what one finds in textbooks (e.g., [3, Theorem 4.41]), and then formally in the language of MLTT.

6.1 Informal proof

(\Rightarrow) *Every equational class is a variety.* Indeed, suppose \mathcal{K} is an equational class axiomatized by term identities \mathcal{E} ; that is, $\mathbf{A} \in \mathcal{K}$ iff $\mathbf{A} \models \mathcal{E}$. Since the classes $\mathbf{H} \mathcal{K}$, $\mathbf{S} \mathcal{K}$, $\mathbf{P} \mathcal{K}$ and \mathcal{K} all satisfy the same set of equations, we have $\mathbf{V} \mathcal{K} \models \mathcal{E}$ for all $(\mathbf{p}, \mathbf{q}) \in \mathcal{E}$, so $\mathbf{V} \mathcal{K} \subseteq \mathcal{K}$.

(\Leftarrow) *Every variety is an equational class.*¹² Let \mathcal{K} be an arbitrary variety. We will describe a set of equations that axiomatizes \mathcal{K} . A natural choice is to take $\mathbf{Th} \mathcal{K}$ and try to prove that $\mathcal{K} = \mathbf{Mod} (\mathbf{Th} \mathcal{K})$. Clearly, $\mathcal{K} \subseteq \mathbf{Mod} (\mathbf{Th} \mathcal{K})$. To prove the converse inclusion, let $\mathbf{A} \in \mathbf{Mod} (\mathbf{Th} \mathcal{K})$. It suffices to find an algebra $\mathbf{F} \in \mathbf{S} (\mathbf{P} \mathcal{K})$ such that \mathbf{A} is a homomorphic image of \mathbf{F} , as this will show that $\mathbf{A} \in \mathbf{H} (\mathbf{S} (\mathbf{P} \mathcal{K})) = \mathcal{K}$.

Let X be such that there exists a surjective environment $\rho : X \rightarrow \mathbb{U}[\mathbf{A}]$.¹³ By the *lift-hom* lemma, there is an epimorphism $h : \mathbf{T} X \rightarrow \mathbb{U}[\mathbf{A}]$ that extends ρ . Put $\mathbb{F}[X] := \mathbf{T} X / \approx$ and let $g : \mathbf{T} X \rightarrow \mathbb{F}[X]$ be the natural epimorphism with kernel \approx . We claim $\ker g \subseteq \ker h$. If the claim is true, then there is a map $f : \mathbb{F}[X] \rightarrow \mathbf{A}$ such that $f \circ g = h$, and since h is surjective so is f . Therefore, $\mathbf{A} \in \mathbf{H} (\mathbb{F} X) \subseteq \mathbf{Mod} (\mathbf{Th} \mathcal{K})$ completing the proof.

It remains to prove the claim $\ker g \subseteq \ker h$. Let u, v be terms and assume $g u = g v$. Since $\mathbf{T} X$ is generated by X , there are terms p, q such that $u = \llbracket \mathbf{T} X \rrbracket p$ and $v = \llbracket \mathbf{T} X \rrbracket q$. Therefore, $\llbracket \mathbb{F}[X] \rrbracket p = g (\llbracket \mathbf{T} X \rrbracket p) = g u = g v = g (\llbracket \mathbf{T} X \rrbracket q) = \llbracket \mathbb{F}[X] \rrbracket q$, so $\mathcal{K} \models p \approx q$; thus, $(p, q) \in \mathbf{Th} \mathcal{K}$. Since $\mathbf{A} \in \mathbf{Mod} (\mathbf{Th} \mathcal{K})$, we obtain $\mathbf{A} \models p \approx q$, which implies that $h u = (\llbracket \mathbf{A} \rrbracket p) \langle \$ \rangle \rho = (\llbracket \mathbf{A} \rrbracket q) \langle \$ \rangle \rho = h v$, as desired.

6.2 Formal proof

(\Rightarrow) *Every equational class is a variety.* We need an arbitrary equational class, which we obtain by starting with an arbitrary collection \mathcal{E} of equations and then defining $\mathcal{K} = \mathbf{Mod} \mathcal{E}$, the class axiomatized by \mathcal{E} . We prove that \mathcal{K} is a variety by showing that $\mathcal{K} = \mathbf{V} \mathcal{K}$. The inclusion $\mathcal{K} \subseteq \mathbf{V} \mathcal{K}$, which holds for all classes \mathcal{K} , is called the *expansive* property of \mathbf{V} .

```

module _ (K : Pred (Algebra  $\alpha \rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \text{ov } \ell$ )) where
  V-expa : K  $\subseteq$  V  $\ell$  (ov ( $\alpha \sqcup \rho^a \sqcup \ell$ )) K
  V-expa {x = A} kA = A, (A, ( $\top$ , ( $\lambda \_ \rightarrow A$ ), ( $\lambda \_ \rightarrow kA$ ), Goal),  $\leq$ -reflexive), IdHomImage
  where
    open Setoid  $\mathbb{D}[\mathbf{A}]$  using (refl)
    open Setoid  $\mathbb{D}[\prod (\lambda \_ \rightarrow \mathbf{A})]$  using () renaming (refl to refl $\prod$ )
    to $\prod$  :  $\mathbb{D}[\mathbf{A}] \rightarrow \mathbb{D}[\prod (\lambda \_ \rightarrow \mathbf{A})]$ 
    to $\prod$  = record { f =  $\lambda x \_ \rightarrow x$ ; cong =  $\lambda xy \_ \rightarrow xy$  }
    from $\prod$  :  $\mathbb{D}[\prod (\lambda \_ \rightarrow \mathbf{A})] \rightarrow \mathbb{D}[\mathbf{A}]$ 
    from $\prod$  = record { f =  $\lambda x \_ \rightarrow x$  tt; cong =  $\lambda xy \_ \rightarrow xy$  tt }
    Goal :  $\mathbf{A} \cong \prod (\lambda x \rightarrow \mathbf{A})$ 
    Goal = mkiso (to $\prod$ , mkhom refl $\prod$ ) (from $\prod$ , mkhom refl) ( $\lambda \_ \_ \rightarrow \text{refl}$ ) ( $\lambda \_ \_ \rightarrow \text{refl}$ )

```

Observe how \mathbf{A} is expressed as (isomorphic to) a product with just one factor (itself), that is, the product $\prod (\lambda x \rightarrow \mathbf{A})$ indexed over the one-element type \top .

For the inclusion $\mathbf{V} \mathcal{K} \subseteq \mathcal{K}$, recall lemma **V-id1** which asserts that $\mathcal{K} \models p \approx q$ implies $\mathbf{V} \ell \iota \mathcal{K} \models p \approx q$; whence, if \mathcal{K} is an equational class, then $\mathbf{V} \mathcal{K} \subseteq \mathcal{K}$, as we now confirm.

```

module _ { $\ell$  : Level} {X : Type  $\ell$ } { $\mathcal{E}$  : {Y : Type  $\ell$ }  $\rightarrow$  Pred (Term Y  $\times$  Term Y) (ov  $\ell$ )} where
  private K = Mod { $\alpha = \ell$ } {X}  $\mathcal{E}$  - an arbitrary equational class

  EqCl $\Rightarrow$ Var : V  $\ell$  (ov  $\ell$ ) K  $\subseteq$  K
  EqCl $\Rightarrow$ Var {A} vA {p} {q} p $\mathcal{E}$ q  $\rho$  = V-id1  $\ell$  {K} {p} {q} ( $\lambda \_ x \tau \rightarrow x$  p $\mathcal{E}$ q  $\tau$ ) A vA  $\rho$ 

```

By **V-expa** and **EqCl \Rightarrow Var**, every equational class is a variety.

¹²The proof we present here is based on [3, Theorem 4.41].

¹³Informally, this is done by assuming X has cardinality at least $\max(|\mathbb{U}[\mathbf{A}]|, \omega)$. Later we will see how to construct an X with the required property in type theory.

4:18 A Machine-Checked Proof of Birkhoff's Theorem

(\Leftarrow) *Every variety is an equational class.* To fix an arbitrary variety, start with an arbitrary class \mathcal{K} of S -algebras and take the *variety closure*, $\mathbf{V} \mathcal{K}$. We prove that $\mathbf{V} \mathcal{K}$ is precisely the collection of algebras that model $\mathbf{Th}(\mathbf{V} \mathcal{K})$; that is, $\mathbf{V} \mathcal{K} = \mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K}))$. The inclusion $\mathbf{V} \mathcal{K} \subseteq \mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K}))$ is a consequence of the fact that $\mathbf{Mod} \mathbf{Th}$ is a closure operator.

```

module _ (K : Pred(Algebra  $\alpha \rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \text{ov } \ell$ )) {X : Type ( $\alpha \sqcup \rho^a \sqcup \ell$ )} where
  private c =  $\alpha \sqcup \rho^a \sqcup \ell$  ;  $\iota = \text{ov } c$ 

  ModTh-closure :  $\mathbf{V} \{ \beta = \beta \} \{ \rho^b \} \{ \gamma \} \{ \rho^c \} \{ \delta \} \{ \rho^d \} \ell \iota \mathcal{K} \subseteq \mathbf{Mod} \{ X = X \} (\mathbf{Th} (\mathbf{V} \ell \iota \mathcal{K}))$ 
  ModTh-closure {x = A} vA {p} {q} x  $\rho = x \mathbf{A} \text{ vA } \rho$ 

```

Our proof of the inclusion $\mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K})) \subseteq \mathbf{V} \mathcal{K}$ is carried out in two steps.

1. Prove $\mathbb{F}[X] \leq \mathbf{C} X$.
2. Prove that every algebra in $\mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K}))$ is a homomorphic image of $\mathbb{F}[X]$.

From 1 we have $\mathbb{F}[X] \in \mathbf{S}(\mathbf{P} \mathcal{K})$, since $\mathbf{C} X$ is a product of algebras in \mathcal{K} . From this and 2 will follow $\mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K})) \subseteq \mathbf{H}(\mathbf{S}(\mathbf{P} \mathcal{K})) (= \mathbf{V} \mathcal{K})$, as desired.

- 1. To prove $\mathbb{F}[X] \leq \mathbf{C} X$, we construct a homomorphism from $\mathbb{F}[X]$ to $\mathbf{C} X$ and then show it is injective, so $\mathbb{F}[X]$ is (isomorphic to) a subalgebra of $\mathbf{C} X$.

```

open FreeHom { $\ell = \ell$ } {K}
open FreeAlgebra K using (homC ;  $\mathbb{F}[\_]$  ; C)
homFC : hom  $\mathbb{F}[X]$  (C X)
homFC = fromHomIm (homC X)

monFC : mon  $\mathbb{F}[X]$  (C X)
monFC = | homFC | , record { isHom = || homFC ||
                           ; isInjective =  $\lambda \{x\}\{y\} \rightarrow \text{fromIm-inj} \mid \text{homC } X \mid \{x\}\{y\}$  }

F≤C :  $\mathbb{F}[X] \leq \mathbf{C} X$ 
F≤C = mon→≤ monFC

open FreeAlgebra K using ( J )

SPF :  $\mathbb{F}[X] \in \mathbf{S} \iota (\mathbf{P} \ell \iota \mathcal{K})$ 
SPF = C X , ((J X) , (|_| , (( $\lambda i \rightarrow \text{fst} \mid i \mid$ ) ,  $\cong\text{-refl}$ ))) , F≤C

```

- 2. Every algebra in $\mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K}))$ is a homomorphic image of $\mathbb{F}[X]$. Indeed,

```

module _ {K : Pred(Algebra  $\alpha \rho^a$ ) ( $\alpha \sqcup \rho^a \sqcup \text{ov } \ell$ )} where
  private c =  $\alpha \sqcup \rho^a \sqcup \ell$  ;  $\iota = \text{ov } c$ 

  Var⇒EqCl :  $\forall \mathbf{A} \rightarrow \mathbf{A} \in \mathbf{Mod}(\mathbf{Th}(\mathbf{V} \ell \iota \mathcal{K})) \rightarrow \mathbf{A} \in \mathbf{V} \ell \iota \mathcal{K}$ 
  Var⇒EqCl A ModThA =  $\mathbb{F}[\cup[\mathbf{A}]]$  , (SPF{ $\ell = \ell$ } K , Aim)
  where
    open FreeAlgebra K using (  $\mathbb{F}[\_]$  )
    epiFIA : epi  $\mathbb{F}[\cup[\mathbf{A}]]$  (Lift-Alg A  $\iota \iota$ )
    epiFIA = F-ModTh-epi-lift{ $\ell = \ell$ }  $\lambda \{p\} \{q\} \rightarrow \text{ModThA}\{p = p\}\{q\}$ 

     $\varphi$  : Lift-Alg A  $\iota \iota$  IsHomImageOf  $\mathbb{F}[\cup[\mathbf{A}]]$ 
     $\varphi$  = epi→ontohom  $\mathbb{F}[\cup[\mathbf{A}]]$  (Lift-Alg A  $\iota \iota$ ) epiFIA

    Aim : A IsHomImageOf  $\mathbb{F}[\cup[\mathbf{A}]]$ 
    Aim =  $\circ\text{-hom} \mid \varphi \mid (\text{from Lift-}\cong) , \circ\text{-IsSurjective } \_ \_ \mid \varphi \mid (\text{fromIsSurjective}(\text{Lift-}\cong\{\mathbf{A} = \mathbf{A}\}))$ 

```

By $\mathbf{ModTh-closure}$ and $\mathbf{Var} \Rightarrow \mathbf{EqCl}$, we have $\mathbf{V} \mathcal{K} = \mathbf{Mod}(\mathbf{Th}(\mathbf{V} \mathcal{K}))$ for every class \mathcal{K} of S -algebras. Thus, every variety is an equational class.

This completes the formal proof of Birkhoff's variety theorem. ◀

7 Conclusion

7.1 Discussion

How do we differ from the classical, set-theoretic approach? Most noticeable is our avoidance of all *size* issues. By using universe levels and level polymorphism, we always make sure we are in a *large enough* universe. So we can easily talk about “all algebras such that . . .” because these are always taken from a bounded (but arbitrary) universe.

Our use of setoids introduces nothing new: all the equivalence relations we use were already present in the classical proofs. The only “new” material is that we have to prove that functions respect those equivalences.

Our first attempt to formalize Birkhoff’s theorem was not sufficiently careful in its handling of variable symbols X . Specifically, this type was unconstrained; it is meant to represent the informal notion of a “sufficiently large” collection of variable symbols. Consequently, we postulated surjections from X onto the domains of all algebras in the class under consideration. But then, given a signature S and a one-element S -algebra \mathbf{A} , by choosing X to be the empty type \perp , our surjectivity postulate gives a map from \perp onto the singleton domain of \mathbf{A} . (For details, see the `Demos.ContraX` module which constructs the counterexample in Agda.)

7.2 Related work

There have been a number of efforts to formalize parts of universal algebra in type theory besides ours. The Coq proof assistant, based on the Calculus of Inductive Constructions, was used by Capretta, in [5], and Spitters and Van der Weegen, in [17], to formalize the basics of universal algebra and some classical algebraic structures. In [11] Gunther et al developed what seemed (prior to the `agda-algebras` library) the most extensive library of formalized universal algebra to date. Like `agda-algebras`, [11] is based on dependent type theory, is programmed in Agda, and goes beyond the basic isomorphism theorems to include some equational logic. Although their coverage is less extensive than that of `agda-algebras`, Gunther et al do treat *multi-sorted* algebras, whereas `agda-algebras` is currently limited to single-sorted structures.

As noted by Abel [1], Amato et al, in [2], have formalized multi-sorted algebras with finitary operators in UniMath. The restriction to finitary operations was due to limitations of the UniMath type theory, which does not have W -types nor user-defined inductive types. Abel also notes that Lyngø and Spitters, in [14], formalize multi-sorted algebras with finitary operators in *Homotopy type theory* ([16]) using Coq [23]. HoTT’s higher inductive types enable them to define quotients as types, without the need for setoids. Lyngø and Spitters prove three isomorphism theorems concerning subalgebras and quotient algebras, but do not formalize universal algebras nor varieties. Finally, in [1], Abel gives a new formal proof of the soundness and completeness theorem for multi-sorted algebraic structures.

References

- 1 Andreas Abel. Birkhoff’s Completeness Theorem for multi-sorted algebras formalized in Agda. *CoRR*, abs/2111.07936, 2021. [arXiv:2111.07936](#).
- 2 Gianluca Amato, Marco Maggesi, and Cosimo Perini Brogi. Universal Algebra in UniMath. *CoRR*, abs/2102.05952, 2021. [arXiv:2102.05952](#).
- 3 Clifford Bergman. *Universal Algebra: fundamentals and selected topics*, volume 301 of *Pure and Applied Mathematics (Boca Raton)*. CRC Press, Boca Raton, FL, 2012.

- 4 G Birkhoff. On the structure of abstract algebras. *Proceedings of the Cambridge Philosophical Society*, 31(4):433–454, October 1935.
- 5 Venanzio Capretta. Universal Algebra in Type Theory. In *Theorem proving in higher order logics (Nice, 1999)*, volume 1690 of *Lecture Notes in Comput. Sci.*, pages 131–148. Springer, Berlin, 1999. doi:10.1007/3-540-48256-3_10.
- 6 William DeMeo. The Agda Universal Algebra Library. GitHub.com, 2020. Ver. 1.0.0. Source code: gitlab.com/ualib/ualib.gitlab.io.
- 7 William DeMeo and Jacques Carette. A Machine-checked Proof of Birkhoff's Variety Theorem in Martin-Löf Type Theory. *CoRR*, abs/2101.10166, 2021. Source code: github.com/ualib/agda-algebras/. doi:10.48550/ARXIV.2101.10166.
- 8 William DeMeo and Jacques Carette. The Agda Universal Algebra Library (agda-algebras). GitHub.com, 2021. Ver. 2.0.1. Source code: [agda-algebras-v.2.0.1.zip](https://github.com/ualib/agda-algebras). Documentation: ualib.org. GitHub repo: github.com/ualib/agda-algebras. doi:10.5281/zenodo.5765793.
- 9 Martín Hötzel Escardó. Introduction to Univalent Foundations of mathematics with Agda. <https://www.cs.bham.ac.uk/~mhe/HoTT-UF-in-Agda-Lecture-Notes/>, May 2019. Accessed on 30 Nov 2021.
- 10 Martín Hötzel Escardó. Introduction to Univalent Foundations of mathematics with Agda. *CoRR*, abs/1911.00580, 2019. arXiv:1911.00580.
- 11 Emmanuel Gunther, Alejandro Gadea, and Miguel Pagano. Formalization of Universal Algebra in Agda. *Electronic Notes in Theoretical Computer Science*, 338:147–166, 2018. The 12th Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2017). doi:10.1016/j.entcs.2018.10.010.
- 12 Jason Z. S. Hu and Jacques Carette. Formalizing Category Theory in Agda. In *Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2021*, pages 327–342, New York, NY, USA, 2021. Association for Computing Machinery. doi:10.1145/3437992.3439922.
- 13 Artur Kornilowicz. Birkhoff theorem for many sorted algebras, 1999.
- 14 Andreas Lyngé and Bas Spitters. Universal Algebra in HoTT. In *Proceedings of the 25th International Conference on Types for Proofs and Programs (TYPES 2019)*, 2019. URL: http://www.ii.uib.no/~bezem/abstracts/TYPES_2019_paper_7.
- 15 Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.
- 16 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Lulu and The Univalent Foundations Program, Institute for Advanced Study, 2013. URL: <https://homotopytypetheory.org/book>.
- 17 Bas Spitters and Eelis Van der Weegen. Type classes for mathematics in type theory. *CoRR*, abs/1102.1323, 2011. arXiv:1102.1323.
- 18 The Agda Team. *Agda Language Reference*, 2021. URL: <https://agda.readthedocs.io/en/v2.6.1/language/index.html>.
- 19 The Agda Team. *Agda Language Reference section on Axiom K*, 2021. URL: <https://agda.readthedocs.io/en/v2.6.1/language/without-k.html>.
- 20 The Agda Team. *Agda Language Reference section on Safe Agda*, 2021. URL: <https://agda.readthedocs.io/en/v2.6.1/language/safe-agda.html#safe-agda>.
- 21 The Agda Team. *The Agda Standard Library*, 2021. URL: <https://github.com/agda/agda-stdlib>.
- 22 The Agda Team. *Agda Tools Documentation section on Pattern matching and equality*, 2021. URL: <https://agda.readthedocs.io/en/v2.6.1/tools/command-line-options.html#pattern-matching-and-equality>.
- 23 The Coq Development Team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0. URL: <http://coq.inria.fr>.

A Imports from the Agda Standard Library

We import a number of definitions from Agda’s standard library (ver. 1.7), as shown below. Notice that these include some adjustments to “standard” Agda syntax; in particular, we use `Type` in place of `Set`, the infix long arrow symbol, `__→__`, in place of `Func` (the type of “setoid functions,” discussed in §2.3 below), and the symbol `__$` in place of `f` (application of the map of a setoid function); we use `fst` and `snd`, and sometimes `|_` and `||_||`, to denote the first and second projections out of the product type `__×__`.

```

– Import 16 definitions from the Agda Standard Library.
open import Data.Unit.Polymorphic   using ( ⊤ ; tt                               )
open import Function                using ( id ; _o_ ; flip                          )
open import Level                   using ( Level                                  )
open import Relation.Binary         using ( Rel ; Setoid ; IsEquivalence            )
open import Relation.Binary.Definitions using ( Reflexive ; Symmetric ; Transitive ; Sym ; Trans )
open import Relation.Binary.PropositionalEquality using ( _≡_                          )
open import Relation.Unary          using ( Pred ; _⊆_ ; _∈_                          )

– Import 23 definitions from the Agda Standard Library and rename 12 of them.
open import Agda.Primitive renaming ( Set to Type ) using ( _⊔_ ; lsuc                )
open import Data.Product  renaming ( proj₁ to fst   ) using ( _×_ ; _,_ ; Σ ; Σ-syntax          )
                           renaming ( proj₂ to snd   )
open import Function      renaming ( Func to __→__ ) using (                               )
open   __→__              renaming ( f to _$ ) using ( cong                               )
open   IsEquivalence      renaming ( refl to refle )
                           renaming ( sym to syme )
                           renaming ( trans to transe ) using (                               )
open   Setoid             renaming ( refl to refls )
                           renaming ( sym to syms )
                           renaming ( trans to transs )
                           renaming ( _≈_ to _≈s_ ) using ( Carrier ; isEquivalence      )

– Assign handles to 3 modules of the Agda Standard Library.
import   Function.Definitions          as FD
import   Relation.Binary.PropositionalEquality as ≡
import   Relation.Binary.Reasoning.Setoid as SetoidReasoning

private variable α ρa β ρb γ ρc δ ρd ρ χ ℓ : Level ;   Γ Δ : Type χ

```


Principal Types as Lambda Nets

Pietro Di Gianantonio   

University of Udine, Italy

Marina Lenisa   

University of Udine, Italy

Abstract

We show that there are connections between *principal type schemata*, *cut-free λ -nets*, and *normal forms* of the λ -calculus, and hence there are correspondences between the normalisation algorithms of the above structures, *i.e.* *unification* of principal types, *cut-elimination* of λ -nets, and *normalisation* of λ -terms. Once the above correspondences have been established, properties of the typing system, such as *typability*, *subject reduction*, and *inhabitation*, can be derived from properties of λ -nets, and vice-versa. We illustrate the above pattern on a specific type assignment system, we study principal types for this system, and we show that they correspond to λ -nets with a non-standard notion of cut-elimination. Properties of the type system are then derived from results on λ -nets.

2012 ACM Subject Classification Theory of computation \rightarrow Lambda calculus; Theory of computation \rightarrow Type structures; Theory of computation \rightarrow Linear logic

Keywords and phrases Lambda calculus, Principal types, Linear logic, Lambda nets, Normalization, Cut elimination

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.5

Funding Work supported by the Italian MUR project PRIN 2017FTXR7S “IT-MaTTeRS” (Methods and Tools for Trustworthy Smart Systems).

Acknowledgements We thank Beniamino Accattoli, Paolo Coppola, Furio Honsell, Ivan Scagnetto, and Gabriele Vanoni for helpful discussions on the subject. We are indebted to the anonymous referees for their valuable comments on the present work.

1 Introduction

The objective of this paper is to present the connections existing among *principal type schemata*, *cut-free λ -nets*, and *normal forms* of λ -calculus. As a consequence of these correspondences there exist correspondences among the normalisation algorithms in the above structures, that is: *unification* of principal type schemata, *cut-elimination* on λ -nets, *normalisation* on λ -terms. While the connection between the two latter is well-known, the relationships between unification of principal type schemata and cut-elimination on λ -nets have rarely been presented explicitly. There are very few works in the literature, [16, 25, 23], based on the above correspondence, and a complete detailed presentation relating also cut-elimination and unification algorithm is missing. We think that it is worthwhile to analyse in detail such a connection since it allows to derive several properties on the type system from properties of λ -nets, so giving new proves and explanations of some already known results concerning type assignment systems. In general, the correspondence existing among types, principal types, λ -nets, and normal forms can be expressed by the following chain of equivalences. For any closed λ -term M :

- M has type τ in the type assignment system, $\vdash M : \tau$, *if and only if*
- M has principal type schema τ' in the principal type assignment system, $\Vdash M : \tau'$, and τ is an instance of τ' , *if and only if*



© Pietro Di Gianantonio and Marina Lenisa;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 5; pp. 5:1–5:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- the λ -net associated with M reduces, by a cut-elimination strategy C , to a cut-free λ -net t , which is the translation of τ' , *if and only if*
- M reduces according to the strategy S to a normal form N , and t is the λ -net associated with N .

Once the above correspondences have been established, it is possible to derive properties concerning the type system and the reduction strategy S from properties of λ -nets and the cut-elimination strategy C , or vice-versa.

In this paper, we present in detail an instance of the general pattern outlined above. Namely, we choose a particular version of type assignment system, we define a corresponding type assignment system for principal types, and we consider a corresponding notion of λ -net.

The type system that we consider includes an intersection operation \wedge on types, which is commutative and associative, but non-idempotent. However, most of our results hold also in the case of idempotency. The type system is almost standard, the only minor non-standard feature is a bang operator used to mark the type subterms on which the \wedge operator can be applied. Then we introduce principal types (type schemata) together with a type assignment system assigning type schemata to λ -terms. Type schemata differ from simple-types by the introduction of type variables and by a box operator, used to mark parts of the type that can be replicated. From type schemata, by suitable instantiation, one gets (ground) types. Each λ -term turns out to have at most one principal type schema, while the set of ground types assignable to a term consists of all the instances of the principal type schema. In the type assignment system for principal types, the rule for introducing the type schema of an application makes use of a generalised *unification algorithm* on types.

We will show that principal types can be seen as an alternative representation of cut-free λ -nets, and the unification algorithm on principal types can be seen as a reformulation of the cut-elimination algorithm for λ -nets. In order to formalise these correspondences, we present a translation from principal type schemata to λ -nets. However, we need to consider a variation of the standard cut-elimination procedure on λ -nets, whereby the cut-rule concerning the conclusions of a weakening and a promotion does not eliminate the box, but leaves a pending term. With this variation the cut-elimination procedure becomes *perpetual*, that is a cut is eliminated only if the λ -net is strongly normalising. On the other hand, we present, in our setting, the standard translation of λ -terms into λ -nets [20, 1]. The main result of this paper can be seen as a commutation result, namely: given a λ -term M , we assign to it a principal type τ , this, in turn, induces a cut-free λ -net, which can be alternatively obtained by applying to M the standard transformation into λ -nets and then the cut-elimination procedure.

As consequences of the correspondences that we have established, we derive the following results on the type assignment system: the *typable terms* are exactly the *strongly normalising* ones, *subject reduction* holds up-to a suitable relation on types, the *inhabitation* problem is decidable.

The present paper builds on [11], where the analogies *involutions as principal types* and *application as unification* have been introduced and explored for a type assignment system related to Abramsky's Geometry of Interaction model of partial involutions.

Related work

Duquesne and Van De Wiele have first described the connection between principal types and proof-nets, [16]. This approach has been further extended by Regnier in his PhD thesis [25]. Similarly to Duquesne, Van De Wiele and Regnier's approach (referred to as the *DVR*

approach in the following), we use indexes to delimit the parts of a type that need to be replicated. However, there exists a long list of differences. Principal types in the DVR approach are pairs composed by a variable and a set of pairs of terms that need to be unified; the most general unifier (MGU) of this set of pairs, if existing, will transform the variable in a principal type in our sense. As a result, principal types in the DVR approach are almost a direct translation of proof-nets, while our principal types describe the normal forms of the proof-nets associated with lambda terms. In defining the principal type of an application, we explicitly define a notion of MGU among two types, while the MGU notion is not present in DVR. As far as this aspect is concerned, we are in the tradition of the Hindley-Milner algorithm for principal types, even if our MGU algorithm is a substantial extension of the original one. In DVR approach, any lambda-term has a principal type, and the type system characterises weakly normalisable terms, while our system just the strongly normalising ones.

Our work is also related to [9, 10, 21, 23]. Our type assignment system for principal types is quite similar to Sistem I, described in [21] and Sistem E, described in [9, 10]. Still, our presentation is completely different, *e.g.* we use a different syntax and different auxiliary procedures; what we call *index variable*, there is called *expansion variable*, what we call duplication, there is called expansion. Moreover, the implementations of duplication and expansion are different, using different auxiliary structures. Nevertheless, one can find similarities in the technique used in System E to describe the scope of expansion (duplication). A connection between the principal types in [21] and interaction-nets is then presented in [23]. However, again the notation is completely different from ours; [21] does not consider standard connectives of MELL but the ones of interaction nets, croissant, sharing, application, . . . , the formulas associated to wires are not MELL proposition, but triples formed by a type, a substitution, and an expansion. Comparing these works to ours, we can say that our principal type algorithm and the connection between types and proof-nets are simpler and more direct. Essentially, our algorithm for principal types is directly inspired by the cut-elimination algorithm on proof-nets. Consequently, the correspondence between the two becomes almost immediate, but the presentation of the principal type algorithm becomes simpler. Moreover, compared to [23], we consider a different and broader set of applications of the correspondence mentioned above. From another point of view, a strict relation between type inference and β -reduction, on so indirectly to λ -nets, is established in [8].

There exists an extensive literature on principal types in general, or more specifically in combination with intersection types, as in our case, [7, 15, 24, 28]. However, the objective of these works is quite different from ours, namely, often principal types are used to define type inference algorithms for simple programming languages. In other works, [12, 26], the connection between principal types and β -normal forms of terms is investigated. Moreover, the type syntax and the type inference algorithm are quite different from ours, and there is no explicit connection to proof-nets.

Non-idempotent intersection types also have a rich bibliography, their interest lies in connection with linear logic, and the possibility to give a finer description of the behaviour lambda terms, *e.g.* [4, 13]. A quite complete presentation of works on this subject can be found in [7]. In this paper, we are going to present a series of new proofs of some results that can be found also in [3, 6, 7, 22].

Summary

In Section 2, we introduce a type assignment system for the λ -calculus, where types include a \wedge and a $!$ operator. In Section 3, we introduce a type assignment system for assigning principal type schemata, and we show how the two systems are related via suitable transformations.

5:4 Principal Types as Lambda Nets

In Section 4, we present λ -nets together with a translation of λ -terms into λ -nets with a modified cut-elimination procedure, and we relate this translation to the standard one and to principal types. In Section 5, we show how the correspondence results established in the previous sections can be used to prove properties of the ground typing system by exploiting results on λ -nets. Finally, in Section 6, final comments and lines for future work appear.

2 The intersection types

We separate the set of types in two sets, *simple-types*, that are used to assign types to subterms in functional position, and *and-types*, used to assign types to subterms that are arguments of a function.

► **Definition 1** (Ground Types). *We introduce the following two sets of types:*

$$\begin{aligned} (\text{SimTypeG } \ni) \sigma, \tau &::= \mathbf{q} \mid \hat{\sigma} \multimap \tau \\ (\text{AndTypeG } \ni) \hat{\sigma}, \hat{\tau} &::= !\tau \mid \hat{\sigma} \wedge \hat{\tau} \end{aligned}$$

where \mathbf{q} ranges over a set of type constants $T\text{Const}$.

Let $\text{TypeG} = \text{SimTypeG} \cup \text{AndTypeG}$, and let μ, ν, ρ range over TypeG .

We assume \wedge to be commutative and associative, i.e. we (implicitly) consider the least congruence on types induced by the following relation:

$$\hat{\sigma}_1 \wedge \hat{\sigma}_2 = \hat{\sigma}_2 \wedge \hat{\sigma}_1 \quad \hat{\sigma}_1 \wedge (\hat{\sigma}_2 \wedge \hat{\sigma}_3) = (\hat{\sigma}_1 \wedge \hat{\sigma}_2) \wedge \hat{\sigma}_3.$$

As a side remark, the main reason to assume \wedge commutative and associative is to have subject reduction, no other result depends on the algebraic properties of \wedge .

► **Definition 2.** *The intersection type system for the λ -calculus derives judgements $\Delta \vdash M : \mu$, where M is a λ term, $\mu \in \text{TypeG}$ and the environment Δ is a set $x_1 : \hat{\sigma}_1, \dots, x_n : \hat{\sigma}_n$, where $\hat{\sigma}_i \in \text{AndTypeG}$ for all i , and the rules for assigning intersection types are the following:*

$$\begin{aligned} &\frac{}{x : !\tau \vdash x : \tau} (\text{ax}) \quad \frac{\Delta \vdash M : \hat{\sigma} \multimap \tau \quad \Delta' \vdash N : \hat{\sigma}}{\Delta \wedge \Delta' \vdash MN : \tau} (\text{app}) \\ &\frac{\Delta, x : \hat{\sigma} \vdash M : \tau}{\Delta \vdash \lambda x.M : \hat{\sigma} \multimap \tau} (\lambda_I) \quad \frac{\Delta \vdash M : \tau \quad x \notin \text{dom}(\Delta) \quad \hat{\sigma} \in \text{AndTypeG}}{\Delta \vdash \lambda x.M : \hat{\sigma} \multimap \tau} (\lambda_A) \\ &\frac{\Delta \vdash M : \tau}{\Delta \vdash M : !\tau} (!) \quad \frac{\Delta \vdash M : \hat{\sigma}_1 \quad \Delta' \vdash M : \hat{\sigma}_2}{\Delta \wedge \Delta' \vdash M : \hat{\sigma}_1 \wedge \hat{\sigma}_2} (\wedge) \end{aligned}$$

where $\Delta \wedge \Delta'$ is the environment Δ'' defined by: $x : \hat{\sigma} \in \Delta''$ if and only if $x : \hat{\sigma} \in \Delta$ and $x \notin \text{dom}(\Delta')$, or $x : \hat{\sigma} \in \Delta'$ and $x \notin \text{dom}(\Delta)$, or $\hat{\sigma} = \hat{\sigma}_1 \wedge \hat{\sigma}_2$ and $x : \hat{\sigma}_1 \in \Delta$ and $x : \hat{\sigma}_2 \in \Delta'$.

3 Principal type schemata

This section aims to show that, for any λ -term M , the set of typing judgements for M are all instances of a unique principal type judgement. In order to obtain the principal schema one needs to introduce in the grammar for types a set of variables and a sort of boxing operators. If a typing judgement is derivable in the principal type assignment system, any ground type judgement obtained by instantiating the type variables and by replicating the parts of the judgement marked by boxing operators will be a valid typing judgement. In the grammar for principal types we mark the positive and negative occurrences of the same variable α by α and $\bar{\alpha}$, respectively:

► **Definition 3.** *The type schemata are defined by the following grammar:*

$$\begin{aligned} (\text{SimType } \exists) \sigma, \tau &::= \alpha \mid \hat{\sigma} \multimap \tau \\ (\text{AndType } \exists) \hat{\sigma}, \hat{\tau} &::= !\tau \mid \hat{\sigma} \wedge \hat{\tau} \mid \Box_i \hat{\sigma} \end{aligned}$$

where $\alpha \in TVar$ is a type variable, and $i \in Ind$ is an index.
Let $Type = \text{SimType} \cup \text{AndType}$, let μ, ν, ρ range over $Type$.

Principal type schemata have been introduced in [24]. Here we present an alternative version of the algorithm for the derivation of principal type schemata that differs in several aspects from the original one. The main difference is that our version of the algorithm is inspired by linear logic and λ -nets, in particular, we introduce a mechanism to define boxes inside type schemata. With these modifications, there exists a direct correspondence between the principal type schemata of a term M and the cut-free normal form of the λ -net representing M . This correspondence goes on by connecting the cut-elimination process on λ -nets and a Most General Unification (MGU) algorithm on principal types schemata. Two principal type schemata are unified by generalised notion of substitution, which we call *substitution-replication*.

► **Definition 4 (Substitution-replication).** *Substitution-replications are transformations on type schemata that are compositions of four basic actions:*

- *Substitution of type variables α by a simple-type σ , denoted by $[\sigma / \alpha]$.*
- *Duplication of all the subtypes appearing under a box operator marked by i , denoted by $Dup(i)$.*
- *Substitution of the box operator index i by a pair of box operators indexed by j_1, j_2 , denoted by $[j_1, j_2 / i]$.*
- *Elimination of all the box operators with index i , denoted by $[\epsilon / i]$.*

The formal definition of the above operators is given by induction on the structure of the type to which they are applied:

- $[\sigma/\alpha](\alpha) = \sigma$, and in the remaining cases by recursive application on the arguments:
 - $[\sigma/\alpha](op_b(\mu_1, \mu_2)) = op_b([\sigma/\alpha](\mu_1), [\sigma/\alpha](\mu_2))$ with $op_b \in \{\multimap, \wedge\}$
 - $[\sigma/\alpha](op_u(\mu)) = op_u([\sigma/\alpha](\mu))$, with $op_u \in \{\Box_j\}_j \cup \{!\}$
 - $[\sigma/\alpha](op_n) = op_n$ with $op_n \in \{\beta \mid \beta \neq \alpha\}$;
- $Dup(i)(\Box_i \hat{\sigma}) = VerL(\Box_i \hat{\sigma}) \wedge VerR(\Box_i \hat{\sigma})$, and in the remaining cases by recursive application on the arguments. In turn, $VerL$ and $VerR$ are defined by induction on their type schemata arguments as:
 - $VerL(\alpha) = \alpha_l, \quad VerR(\alpha) = \alpha_r,$
 - $VerL(\Box_j \mu) = \Box_{j_l}(VerL(\mu)), \quad VerR(\Box_j \mu) = \Box_{j_r}(VerR(\mu)),$
 - and in the remaining cases by recursive application on the arguments;
- $[j_1, j_2 / i](\Box_i \hat{\sigma}) = \Box_{j_1} \Box_{j_2} \hat{\sigma}$, and in the remaining cases by recursive application on the arguments;
- $[\epsilon / i](\Box_i \hat{\sigma}) = \hat{\sigma}$, and in the remaining cases by recursive application on the arguments.

According to the above definition, the duplication operator $Dup(i)$ replaces a subtype in the form $\Box_i \hat{\sigma}$ by $VerL(\Box_i \hat{\sigma}) \wedge VerR(\Box_i \hat{\sigma})$, where $VerL(\Box_i \hat{\sigma})$ and $VerR(\Box_i \hat{\sigma})$ are distinct copies of $\Box_i \hat{\sigma}$, the left and the right versions. These two new versions have the same syntactic structure but use two distinct sets of indexes and type variables. To accomplish this we assume that to each index i and type variable α are associated two new fresh instances, their left and their right versions, denoted by i_l and i_r , α_l and α_r , respectively.

The operations of duplication, substitution of a box index by a pair of box indexes, and elimination of a box index correspond to box duplication, box insertion rule, and elimination of a box boundary in cut-elimination of proof nets.

The role of the indexes in the box operators in the types is to mark explicitly the parts of a type that can be replicated, and this, in turn, is fundamental to have a principal type whose instantiations define the whole set of types associated with a term.

It is possible to extend to type schemata and substitution-replications the definitions of partial order and MGU on terms and substitutions:

► **Definition 5.**

- (i) On type schemata we define the subsumption preorder \preceq :
 $\mu \preceq \nu$ if and only if there exists a substitution-replication T such that $\mu = T(\nu)$.
- (ii) The generality preorder on substitution-replications is defined by:
 $S \preceq T$ iff there exists a substitution-replication R such that $S = R \circ T$.
- (iii) Two type schemata, μ, ν , have a unifier U if U is a substitution-replication such that $U(\mu) = U(\nu)$.

The equivalence relation on type schemata induced by the subsumption preorder can be characterised as the least equivalence relation closed by renaming of type variables and indexes, and by uniformly substituting all the boxes with given index i , \square_i , by a pair of boxes $\square_{j_1} \square_{j_2}$ with two fresh indexes j_1, j_2 . Namely, $\square_i \tau = \square_{j_1} \square_{j_2} \tau$, because $[j_1, j_2/i](\square_i \tau) = \square_{j_1} \square_{j_2} \tau$ and $[\epsilon/j_1] \circ [\epsilon/j_2] \circ [i, j_1/j_1](\square_{j_1} \square_{j_2} \tau) = \square_i \tau$.

► **Proposition 6.**

- (i) Any two type schemata μ and ν have a supremum ρ in the preorder \preceq .
- (ii) If two type schemata, μ, ν , have a unifier, then they have a most general unifier, MGU.

On type schemata we define a partial algorithm that, if converging, finds the MGU of two types. In order to study some regularities of our typing system, it is convenient to introduce the following definition, where we distinguish between positive and negative type schemata:

► **Definition 7.** Let $PSimType$, $NSimType$, $PAndType$, $NAndType$ be the sets of type schemata defined by the following grammars:

- $(PSimType \ni) \tau ::= \alpha \mid \widehat{\sigma} \multimap \tau \quad (PAndType \ni) \widehat{\tau} ::= \square_i! \tau$
- $(NSimType \ni) \sigma ::= \bar{\alpha} \mid \widehat{\tau} \multimap \sigma \quad (NAndType \ni) \widehat{\sigma} ::= !\sigma \mid \square_i \widehat{\sigma} \mid \widehat{\sigma}_1 \wedge \widehat{\sigma}_2$
- Let $PType = PSimType \cup PAndType$, let μ range over $PType$.
- Let $NType = NSimType \cup NAndType$, let ν range over $NType$.

Positive types are assigned to λ -terms. Inside a positive type, the subterms occurring in negative positions, i.e. on the left of an odd number of \multimap , will be negative types, while those occurring in positive positions will be positive. A positive type describes the behaviour of a λ -term or the behaviour of a subterm inside a normalised λ -term, while a negative type describes how a potential argument of a λ -term is used inside the term. This fact explains the different shapes of positive and negative types. For example, when an \wedge -negative type is the meet of several component types, the corresponding argument is used several times inside a term, in different contexts, and each use of the argument is described by a component type.

Notice that, in particular, in the above definition we distinguish between positive and negative occurrences of type variables.

The MGU algorithm that we introduce below is defined on pairs of types, (ν, μ) , where $\mu \in PType$, $\nu \in NType$, and μ, ν are either both simple or both “and” types. As we will see, this is sufficient for our purposes.

► **Definition 8.** Let μ, ν be a positive and a negative type, respectively, with μ, ν either both simple or both and-types. The partial algorithm $MGU(\nu, \mu)$ yields a substitution-replication U on types and indexes such that $U(\nu) = U(\mu)$, and it is defined via the following rules:

$$\frac{\alpha \notin \text{Var}(\sigma)}{MGU(\sigma, \alpha) = [\sigma / \bar{\alpha}]} \quad \frac{\alpha \notin \text{Var}(\tau)}{MGU(\bar{\alpha}, \tau) = [\tau / \alpha]}$$

$$\frac{MGU(\hat{\sigma}_1, \hat{\tau}_1) = U_1 \quad MGU(U_1(\sigma_2), U_1(\tau_2)) = U_2}{MGU(\hat{\tau}_1 \multimap \sigma_2, \hat{\sigma}_1 \multimap \tau_2) = U_2 \circ U_1}$$

$$\frac{MGU(\hat{\sigma}_1, \text{VerL}(\Box_i! \tau)) = U_1 \quad MGU(U_1(\hat{\sigma}_2), U_1(\text{VerR}(\Box_i! \tau))) = U_2}{MGU(\hat{\sigma}_1 \wedge \hat{\sigma}_2, \Box_i! \tau) = U_2 \circ U_1 \circ \text{Dup}(i)}$$

$$\frac{MGU(\hat{\sigma}, \Box_i! \tau) = U}{MGU(\Box_j \hat{\sigma}, \Box_i! \tau) = U \circ [j, i / i]} \quad \frac{MGU(\sigma, \tau) = U}{MGU(!\sigma, \Box_i! \tau) = U \circ [\epsilon / i]}$$

One can notice that, to avoid introducing an extra ad hoc rule, the unification of two type schemata of the form $\Box_j! \hat{\sigma}$ and $\Box_i! \tau$ is performed by first duplicating the box i , and then removing one instance of the two boxes which have been just created. It is not difficult to check that the MGU algorithm is well-defined, *i.e.*:

► **Lemma 9.** Let μ, ν be a positive and a negative type, respectively, with μ, ν either both simple or both and-types. Then

- (i) if $MGU(\nu, \mu)$ terminates yielding a substitution-replication U , then U is the most general unifier of μ, ν ;
- (ii) if μ, ν have a unifier, then the MGU algorithm terminates with a well-formed substitution-replication U , *i.e.*, for each variable α , the substitution part includes either $[\sigma / \bar{\alpha}]$ or $[\tau / \alpha]$.

Since the \wedge -operator is commutative and associative, the MGU-algorithm performs different runs starting from different writings of the input values ν, μ . However, since by Proposition 6 the MGU solution is unique up-to the equivalence relation on type schemata of Definition 5, all runs lead to the same result (up-to equivalence relation). A similar consideration applies if one considers an alternative formulation of the MGU algorithm, where the rule for the \multimap case unifies the subterms of the types in a different order. The alternative formulation of the MGU algorithm gives the same results, up-to equivalence relation on type schemata.

When the two types μ, ν do not have a unifier, then the MGU algorithm does not terminate, because there is an infinite number of applications of the rules. This happens when searching for principal type schemata of non-normalising λ -terms, for example the λ -term $\Omega = (\lambda x.xx)(\lambda x.xx)$. Namely, as we will see, to assign a principal type to Ω one needs to find a unifier of the types $!(\Box_i! \alpha \multimap \bar{\beta}) \wedge \Box_i! \bar{\alpha}$ and $\Box_{j'}!(\Box_{i'}! \alpha' \multimap \bar{\beta}') \wedge \Box_{i'}! \bar{\alpha}' \multimap \beta'$. In more detail, the steps of the MGU algorithm for the above case are given in the following example.

► **Example 10.** Let $\hat{\sigma} = !(\Box_i! \alpha \multimap \bar{\beta}) \wedge \Box_i! \bar{\alpha}$. Given a generic list p of l and r labels, we denote by $\hat{\sigma}_p$ the type $!(\Box_{i_p}! \alpha_p \multimap \bar{\beta}_p) \wedge \Box_{i_p}! \bar{\alpha}_p$, where $\alpha_{p_0 \dots p_n}$ denotes the variable $((\alpha_{p_0}) \dots)_{p_n}$ and analogously for the index i_p .

With this notation, the MGU evaluation mentioned above is equivalent to evaluate:

$$\begin{aligned} U &= MGU(\hat{\sigma}_l, \Box_j!(\hat{\sigma}_r \multimap \beta_r)) \\ &= MGU(!(\Box_{i_l}! \alpha_l \multimap \bar{\beta}_l) \wedge \Box_{i_l}! \bar{\alpha}_l, \Box_j!(\hat{\sigma}_r \multimap \beta_r)) = U_2 \circ U_1 \circ \text{Dup}(j) \end{aligned}$$

where

$$\begin{aligned}
 U_1 &= MGU(!(\Box_{i_l}!\alpha_l \multimap \bar{\beta}_l), \Box_{j_r}!(\hat{\sigma}_{r_l} \multimap \beta_{r_l})) \\
 &= MGU(\Box_{i_l}!\alpha_l \multimap \bar{\beta}_l, \hat{\sigma}_{r_l} \multimap \beta_{r_l}) \circ [\epsilon/j_l] \\
 &= MGU(\Box_{i_l}!\alpha_l \multimap \bar{\beta}_l, (!(\Box_{i_{r_l}}!\alpha_{r_l} \multimap \bar{\beta}_{r_l}) \wedge \Box_{i_{r_l}}!\bar{\alpha}_{r_l}) \multimap \beta_{r_l}) \circ [\epsilon/j_l] \\
 &= [\bar{\beta}_l/\bar{\beta}_{r_l}] \circ MGU(!(\Box_{i_{r_l}}!\alpha_{r_l} \multimap \bar{\beta}_{r_l}) \wedge \Box_{i_{r_l}}!\bar{\alpha}_{r_l}, \Box_{i_l}!\alpha_l) \circ [\epsilon/j_l] \\
 &= [\bar{\beta}_l/\bar{\beta}_{r_l}] \circ [\bar{\alpha}_{r_l}/\bar{\alpha}_{l_r}] \circ [\epsilon/i_{l_r}] \circ [i_{r_l}, i_{l_r}/i_{l_r}] \circ [\Box_{i_{r_l}}!\alpha_{r_l} \multimap \bar{\beta}_{r_l}/\bar{\alpha}_{l_l}] \circ [\epsilon/i_{l_l}] \circ Dup(i) \circ [\epsilon/j_l]
 \end{aligned}$$

and

$$U_2 = MGU(U_1(\Box_{i_l}!\bar{\alpha}_l), U_1(\Box_{j_r}!(\hat{\sigma}_{r_r} \multimap \beta_{r_r}))) = MGU([\bar{\beta}_l/\bar{\beta}_{r_l}](\hat{\sigma}_{r_l}), \Box_{j_r}!(\hat{\sigma}_{r_r} \multimap \beta_{r_r})).$$

Up-to renaming of the indexes, U_2 coincides with U , so the attempt to define the unifier U leads to build an infinite sequence of equivalent unification problems, none of them having solution.

► **Definition 11.** *The principal intersection type system for the λ -calculus derives judgments $\Delta \Vdash M : \mu$, where $\mu \in PType$, the environment Δ is a set $x_1 : \hat{\sigma}_1, \dots, x_n : \hat{\sigma}_n$, $\hat{\sigma}_i \in NType$ for all i , and the rules for assigning principal intersection types are the following:*

$$\begin{array}{c}
 \frac{}{x : \bar{\alpha} \Vdash x : \alpha} (ax) \quad \frac{\Delta, x : \hat{\sigma} \Vdash M : \tau}{\Delta \Vdash \lambda x.M : \hat{\sigma} \multimap \tau} (\lambda_l) \quad \frac{\Delta \Vdash M : \tau \quad x, \alpha \text{ fresh}}{\Delta \Vdash \lambda x.M : \bar{\alpha} \multimap \tau} (\lambda_A) \\
 \\
 \frac{\Delta \Vdash M : \alpha \quad \Delta' \Vdash N : \hat{\tau} \quad \beta \text{ fresh}}{[(\hat{\tau} \multimap \bar{\beta}) / \bar{\alpha}](\Delta \wedge \Delta') \Vdash MN : \beta} (appNorm) \quad \frac{\Delta \Vdash M : \tau \quad i \text{ fresh}}{\Box_i \Delta \Vdash M : \Box_i \tau} (box) \\
 \\
 \frac{\Delta \Vdash M : \hat{\sigma} \multimap \tau_1 \quad \Delta' \Vdash N : \hat{\tau} \quad U = MGU(\hat{\sigma}, \hat{\tau})}{U(\Delta \wedge \Delta') \Vdash MN : U(\tau_1)} (appRedex)
 \end{array}$$

where

- in each derivation, for rules with two premises, i.e. $\frac{\Delta_1 \Vdash M_1 : \tau_1 \quad \Delta_2 \Vdash M_2 : \tau_2}{\Delta \Vdash M : \tau}$, we implicitly assume that Δ_1, τ_1 and Δ_2, τ_2 have disjoint sets of type variables and indexes.
- $\Box_i(x_1 : \hat{\sigma}_1, \dots, x_n : \hat{\sigma}_n)$ denotes the environment $x_1 : \Box_i \hat{\sigma}_1, \dots, x_n : \Box_i \hat{\sigma}_n$;
- $\Delta \wedge \Delta'$ is defined as in Definition 2;
- U denotes the most general unifier of types $\hat{\sigma}, \hat{\tau}$, obtained from the MGU algorithm previously defined.

Next one can observe that all derivable judgements have a specific form, for this purpose we define the notions of *well-formed judgement* and *well-formed application* of a substitution-replication:

► **Definition 12.**

- (i) A well-formed judgement $\Delta \Vdash M : \mu$ is a judgement where:
 - each type variable occurs at most twice, at most once as positive simple-type $PSimType$, and once as negative simple-type $NSimType$;
 - each index occurs once as box index in a positive type, $\Box_i \tau$, and an arbitrary number of times as box index of and-types of the shape $\Box_i \hat{\sigma}$.
- (ii) A well-formed application of a substitution-replication, $U(\mu)$, satisfies the following condition: if U contains an action on an index i , that is an operation in the form $Dup(i)$, $[j_1, j_2/i]$, $[\epsilon/i]$, then i appears only negatively in μ , that is the boxes with index i have all form $\Box_i \hat{\sigma}$.

► **Lemma 13.** For every λ -term M :

- (i) there is at most one derivable judgement $\Delta \Vdash M : \tau$, up-to-renaming of type variables and indexes;
- (ii) every derivable judgement $\Delta \Vdash M : \tau$ is well-formed;
- (iii) in deriving $\Delta \Vdash M : \tau$ the substitution-replications obtained via the MGU algorithm induce well-formed applications.

Proof. Item (i) is proved by induction on the structure of M . Items (ii) and (iii) are proved by first proving, by induction on the derivation, that any well-formed application of the MGU algorithm defines a unification that is well-formed, and that any application of the unification is well-formed. After that, item (ii) is proved by induction on the derivation, and item (iii) is almost straightforward. ◀

► **Example 14.** Completing Example 10, one can show that, by applying the rules of Definition 11, we derive the following chain of type assignments:

$$\begin{aligned}
x : !\bar{\gamma} &\Vdash x : \gamma \\
x : \square_i !\bar{\alpha} &\Vdash x : \square_i !\alpha \\
x : !(\square_i !\alpha \multimap \bar{\beta}) \wedge \square_i !\bar{\alpha} &\Vdash xx : \beta \\
&\Vdash \lambda x.xx : !(\square_i !\alpha \multimap \bar{\beta}) \wedge \square_i !\bar{\alpha} \multimap \beta \\
&\Vdash (\lambda x.xx) : \square_j !(\square_i !\alpha \multimap \bar{\beta}) \wedge \square_i !\bar{\alpha} \multimap \beta.
\end{aligned}$$

However, $\Omega = (\lambda x.xx)(\lambda x.xx)$ is not typable, since the types $!(\square_i !\alpha \multimap \bar{\beta}) \wedge \square_i !\bar{\alpha}$ and $\square_j !(\square_{i'} !\alpha' \multimap \bar{\beta}') \wedge \square_{i'} !\bar{\alpha}' \multimap \beta'$ do not have a unifier, as shown in Example 10 above.

A more complex and somewhat classical example, which shows in more detail how duplication works, is obtained by considering the Church numeral two, $2 = \lambda f.\lambda x.f(fx)$, applied to itself, 22 :

► **Example 15.** The principal type of the λ -term 2 is derived by the following chain of principal typing judgements:

$$\begin{aligned}
f : !\bar{\alpha}' &\Vdash f : \alpha' \\
x : \square_j !\bar{\gamma} &\Vdash x : \square_j !\gamma \\
f : !(\square_j !\gamma \multimap \bar{\beta}), x : \square_j !\bar{\gamma} &\Vdash fx : \beta \\
f : \square_i !(\square_j !\gamma \multimap \bar{\beta}), x : \square_i \square_j !\bar{\gamma} &\Vdash (fx) : \square_i !\beta \\
f : !(\square_i !\beta \multimap \bar{\alpha}) \wedge \square_i !(\square_j !\gamma \multimap \bar{\beta}), x : \square_i \square_j !\bar{\gamma} &\Vdash f(fx) : \alpha \\
f : !(\square_i !\beta \multimap \bar{\alpha}) \wedge \square_i !(\square_j !\gamma \multimap \bar{\beta}) &\Vdash \lambda x.f(fx) : \square_i \square_j !\bar{\gamma} \multimap \alpha \\
&\Vdash 2 : !(\square_i !\beta \multimap \bar{\alpha}) \wedge \square_i !(\square_j !\gamma \multimap \bar{\beta}) \multimap (\square_i \square_j !\bar{\gamma} \multimap \alpha)
\end{aligned}$$

To simplify the evaluation, we introduce the following notation: given a list p of l and r labels, we denote by σ_p the type $!(\square_{i_p} !\beta_p \multimap \bar{\alpha}_p) \wedge \square_{i_p} !(\square_{j_p} !\gamma_p \multimap \bar{\beta}_p)$ and by τ_p the type $\sigma_p \multimap (\square_{i_p} \square_{j_p} !\bar{\gamma}_p \multimap \alpha_p)$.

Taking advantage by these notations, up-to renaming of variables and indexes, we have that:

$$\Vdash 22 : U(\square_{i_l} \square_{j_l} !\bar{\gamma}_l \multimap \alpha_l),$$

where $U = MGU(\sigma_l, \square_k !\tau_r)$.

5:10 Principal Types as Lambda Nets

By applying the MGU rules, the evaluation of U develops as follows:

$$U = U_2 \circ U_1 \circ Dup(k)$$

where

$$\begin{aligned} U_1 &= MGU(!(\Box_{i_l}!\beta_l \multimap \bar{\alpha}_l), VerL(\Box_k!\tau_r)) \\ &= MGU(!(\Box_{i_l}!\beta_l \multimap \bar{\alpha}_l), \Box_{k_l}(\sigma_{rl} \multimap \Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl})) \\ &= MGU(\Box_{i_l}!\beta_l \multimap \bar{\alpha}_l, \sigma_{rl} \multimap \Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \circ [\epsilon/k_l]. \end{aligned}$$

It follows $U_1 = U_4 \circ U_3 \circ [\epsilon/k_l]$, where

$$\begin{aligned} U_3 &= MGU(\sigma_{rl}, \Box_{i_l}!\beta_l) \\ &= MGU(!(\Box_{i_{rl}}!\beta_{rl} \multimap \bar{\alpha}_{rl}) \wedge \Box_{i_{rl}}!(\Box_{j_{rl}}!\gamma_{rl} \multimap \bar{\beta}_{rl}), \Box_{i_l}!\beta_l) \\ &= [\gamma_{rl} \multimap \bar{\beta}_{rl}/\bar{\beta}_{lr}] \circ [i_{rl}/i_{lr}] \circ [\Box_{i_{rl}}!\beta_{rl} \multimap \bar{\alpha}_{rl}/\bar{\beta}_{ll}] \circ [\epsilon/i_{ll}] \circ Dup(i_l) \end{aligned}$$

To have a more compact notation, in the above formula, the substitution $[\epsilon/i_{lr}] \circ [i_{rl}, i_{lr}/i_{lr}]$, formally obtained by application of the MGU rules, is written as $[i_{rl}/i_{lr}]$; in the following a similar simplification is used. Carrying on with the evaluation, we have:

$$U_4 = MGU(U_3(\bar{\alpha}_l), U_3(\Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl})) = [\Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}/\bar{\alpha}_l]$$

It is not difficult to check that U_1 is indeed the unifier of the types $!(\Box_{i_l}!\beta_l \multimap \bar{\alpha}_l)$ and $\Box_{k_l}(\sigma_{rl} \multimap \Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl})$.

Continuing with the evaluation, we have:

$$\begin{aligned} U_2 &= MGU(U_1(\Box_{i_l}!(\Box_{j_l}!\gamma_l \multimap \bar{\beta}_l)), U_1(VerR\Box_k!\tau_r)) \\ &= MGU(!(\Box_{j_{ll}}!\gamma_{ll} \multimap \Box_{i_{rl}}!\beta_{rl} \multimap \bar{\alpha}_{rl}) \wedge \Box_{i_{rl}}!(\Box_{j_{lr}}!\gamma_{lr} \multimap \Box_{j_{rl}}!\gamma_{rl} \multimap \bar{\beta}_{rl}), \Box_{k_r}!\tau_{rr}) \\ &= U_6 \circ U_5 \circ Dup(k_r), \text{ where} \end{aligned}$$

$$\begin{aligned} U_5 &= MGU(!(\Box_{j_{ll}}!\gamma_{ll} \multimap \Box_{i_{rl}}!\beta_{rl} \multimap \bar{\alpha}_{rl}), \Box_{k_{rl}}!(\sigma_{rrl} \multimap (\Box_{i_{rrl}}\Box_{j_{rrl}}!\bar{\gamma}_{rrl} \multimap \alpha_{rrl}))) \\ &= [\bar{\alpha}_{rl}/\bar{\alpha}_{rrl}] \circ [\bar{\gamma}_{rrl}/\bar{\beta}_{rl}] \circ [i_{rrl}, j_{rrl}/i_{rl}] \circ [\sigma_{rrl}/\bar{\gamma}_{ll}] \circ [\epsilon/j_{ll}] \circ [\epsilon/k_{rl}], \text{ and} \end{aligned}$$

$$\begin{aligned} U_6 &= MGU(U_5(\Box_{i_{rl}}!(\Box_{j_{lr}}!\gamma_{lr} \multimap \Box_{j_{rl}}!\gamma_{rl} \multimap \bar{\beta}_{rl})), U_5(VerR(\Box_{k_r}!\tau_{rr}))) \\ &= MGU(\Box_{i_{rrl}}\Box_{j_{rrl}}!(\Box_{j_{lr}}!\gamma_{lr} \multimap \Box_{j_{rl}}!\gamma_{rl} \multimap \bar{\gamma}_{rrl}), \\ &\quad (\Box_{k_{rr}}!(\sigma_{rrr} \multimap (\Box_{i_{rrr}}\Box_{j_{rrr}}!\bar{\gamma}_{rrr} \multimap \alpha_{rrr})))) \\ &= [\bar{\gamma}_{rrl}/\bar{\alpha}_{rrr}] \circ [\bar{\gamma}_{rrr}/\bar{\gamma}_{rl}] \circ [i_{rrr}, j_{rrr}/j_{rl}] \circ [\sigma_{rrr}/\bar{\gamma}_{lr}] \circ [\epsilon/j_{lr}] \circ [i_{rrl}, j_{rrl}/k_{rr}] \end{aligned}$$

Summing up we obtain:

$$\begin{aligned} &U(\Box_{i_l}\Box_{j_l}!\bar{\gamma}_l \multimap \alpha_l) \\ &= U_2 \circ U_4 \circ U_3 \circ [\epsilon/k_l] \circ Dup(k)(\Box_{i_l}\Box_{j_l}!\bar{\gamma}_l \multimap \alpha_l) \\ &= U_2 \circ U_4 \circ [\gamma_{rl} \multimap \bar{\beta}_{rl}/\bar{\beta}_{lr}] \circ [i_{rl}/i_{lr}] \circ [\Box_{i_{rl}}!\beta_{rl} \multimap \bar{\alpha}_{rl}/\bar{\beta}_{ll}] \circ [\epsilon/i_{ll}] \circ Dup(i_l)(\Box_{i_l}\Box_{j_l}!\bar{\gamma}_l \multimap \alpha_l) \\ &= U_2 \circ U_4 \circ [\gamma_{rl} \multimap \bar{\beta}_{rl}/\bar{\beta}_{lr}] \circ [i_{rl}/i_{lr}]((\Box_{j_{ll}}!\bar{\gamma}_{ll} \wedge \Box_{i_{rl}}\Box_{j_{lr}}!\bar{\gamma}_{lr}) \multimap \alpha_l) \\ &= U_6 \circ U_5 \circ Dup(k_r)((\Box_{j_{ll}}!\bar{\gamma}_{ll} \wedge \Box_{i_{rl}}\Box_{j_{lr}}!\bar{\gamma}_{lr}) \multimap \Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \\ &= U_6 \circ [\bar{\alpha}_{rl}/\bar{\alpha}_{rrl}] \circ [\bar{\gamma}_{rrl}/\bar{\beta}_{rl}] \circ [i_{rrl}, j_{rrl}/i_{rl}] \circ [\sigma_{rrl}/\bar{\gamma}_{ll}] \circ [\epsilon/j_{ll}] \\ &\quad ((\Box_{j_{ll}}!\bar{\gamma}_{ll} \wedge \Box_{i_{rl}}\Box_{j_{lr}}!\bar{\gamma}_{lr}) \multimap \Box_{i_{rl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \\ &= U_6 \circ [\bar{\alpha}_{rl}/\bar{\alpha}_{rrl}]((\sigma_{rrl} \wedge \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{j_{lr}}!\bar{\gamma}_{lr}) \multimap \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \\ &= [\bar{\beta}_{rl}/\bar{\alpha}_{rrr}] \circ [\bar{\gamma}_{rrr}/\bar{\gamma}_{rl}] \circ [i_{rrr}, j_{rrr}/j_{rl}] \circ [\sigma_{rrr}/\bar{\gamma}_{lr}] \circ [\epsilon/j_{lr}] \\ &\quad ((([\bar{\alpha}_{rl}/\bar{\alpha}_{rrl}]\sigma_{rrl}) \wedge \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{j_{lr}}!\bar{\gamma}_{lr}) \multimap \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{j_{rl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \\ &= [\bar{\gamma}_{rrl}/\bar{\alpha}_{rrr}] \circ [\bar{\gamma}_{rrr}/\bar{\gamma}_{rl}](((\bar{\alpha}_{rl}/\bar{\alpha}_{rrl})\sigma_{rrl}) \wedge \Box_{i_{rrl}}\Box_{j_{rrl}}\sigma_{rrr}) \multimap \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{i_{rrr}}\Box_{j_{rrl}}!\bar{\gamma}_{rl} \multimap \alpha_{rl}) \\ &= ((([\bar{\alpha}_{rl}/\bar{\alpha}_{rrl}]\sigma_{rrl}) \wedge \Box_{i_{rrl}}\Box_{j_{rrl}}([\bar{\gamma}_{rrl}/\bar{\alpha}_{rrr}]\sigma_{rrr})) \multimap \Box_{i_{rrl}}\Box_{j_{rrl}}\Box_{i_{rrr}}\Box_{j_{rrl}}!\bar{\gamma}_{rrr} \multimap \alpha_{rl}) \end{aligned}$$

which coincides with the principal type of the Church number 4.

3.1 Relating the typing systems

The ground typing system and the principal type system are connected via substitution-replications:

► **Theorem 16.**

- (i) If $\Delta \vdash M : \mu$ is derivable, then there exist Δ' , μ' and a substitution-replication T such that $\Delta' \Vdash M : \mu'$ is derivable, $T(\Delta') = \Delta$ and $T(\mu') = \mu$.
- (ii) For any derivable judgement $\Delta \Vdash M : \mu$ and for all substitution-replications T , if the judgement $T(\Delta) \vdash M : T(\mu)$ contains only ground types, then it is derivable.

Proof.

- (i) By induction on \vdash -judgement derivations, using Lemma 13(i) above, and the fact that *MGU* gives the most general substitution-replication (Lemma 9).
- (ii) By induction on \Vdash -judgement derivations. ◀

4 Principal types and lambda nets

The aim of this section is to show that the principal type of a λ -term M is an alternative description of the cut-free form of the λ -net representing M . Moreover, the *MGU* algorithm amounts to the cut-elimination algorithm on λ -nets.

In order to exhibit the correspondence, and maybe against the spirit of proof nets, we are going to give a representation of proof structures by sets of terms. Similar representations have been used in [17] for MLL, and in [18] for interaction nets. The main idea in this representation is to use a pair of variables to represent a proof-net axiom, and a term to represent a proof-net conclusion. Differently from the cited works, we consider MELL, the multiplicative and exponential fragment of LL, so we give a term representation also for boxes.

The following grammar defines a set of *proof terms* in the form $t : A$, where A is a formula of MELL, and t is a term representing the part of a proof structure having A as conclusion:

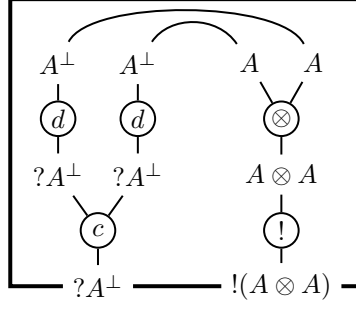
$$\begin{aligned}
 t : A ::= & \alpha : A \mid \bar{\alpha} : A^\perp \mid (t : A) \wp (s : B) : A \wp B \\
 & \mid (t : A) \otimes (s : B) : A \otimes B \mid !_i(t : A) : !A \\
 & \mid \mathbf{d}(t : A) : ?A \mid \mathbf{c}((t : ?A), (s : ?A)) : ?A \mid \mathbf{w} : ?A \\
 & \mid \mathbf{cut}(t : A, s : A^\perp) : \perp \\
 & \mid \square_i(t : ?A) : ?A \mid \square_i(t : \perp) : \perp
 \end{aligned}$$

where α ranges over a set of basic variables $TVar$, and i over a set of indexes Ind . We assume that $TVar$ is equipped with an involution operation $\alpha \mapsto \bar{\alpha}$. The idea is to use a pair $\alpha, \bar{\alpha}$ to mark the pair of formulas introduced by an axiom, and an index i to mark the boundary of a box in proof structures. The constructor \mathbf{c} is taken to be associative and commutative.

In an almost straightforward way, one can transform a proof structure with conclusions A_1, \dots, A_m and containing a set of cuts between the pairs of formulae $(B_1, B_1^\perp), \dots, (B_m, B_m^\perp)$ in a set of proof terms $\vdash t_1 : A_1, \dots, t_m : A_m, \mathbf{cut}(s_1 : B_1, s'_1 : B_1^\perp) : \perp, \dots, \mathbf{cut}(s_m : B_m, s'_m : B_m^\perp) : \perp$.

► **Example 17.** With the above notation, the proof-net represented in the standard notation by the following diagram:

5:12 Principal Types as Lambda Nets



is written as the list of proof terms:

$$\square_i(\mathbf{c}(\mathbf{d}(\bar{\alpha} : A^\perp) : ?A^\perp, \mathbf{d}(\bar{\beta} : A^\perp) : ?A^\perp) : ?A^\perp, !_i((\beta : A) \otimes (\alpha : A) : A \otimes A) : !(A \otimes A).$$

To limit redundancy, the MELL formulae appearing inside a proof term will be most of the times omitted. With this convention, the above formula is written as:

$$\square_i(\mathbf{c}(\mathbf{d}\bar{\alpha}, \mathbf{d}\bar{\beta})) : ?A^\perp, !_i(\beta \otimes \alpha) : !(A \otimes A).$$

The sets of proof terms representing a valid proof structure is inductively defined by the following set of rules:

$$\begin{array}{ccc} \frac{}{\models \bar{\alpha} : A^\perp, \alpha : A} \text{(ax)} & \frac{\models \Gamma, t : A, s : B}{\models \Gamma, t \wp s : A \wp B} \text{(par)} & \frac{\models \Gamma_1, t : A \quad \models \Gamma_2, s : B}{\models \Gamma_1, \Gamma_2, t \otimes s : A \otimes B} \text{(tens)} \\ \frac{\models \Gamma}{\models \Gamma, \mathbf{w} : ?A} \text{(weak)} & \frac{\models \Gamma, t : A}{\models \Gamma, \mathbf{d}t : ?A} \text{(der)} & \frac{\models \Gamma, t_1 : ?A, t_2 : ?A}{\models \Gamma, \mathbf{c}(t_1, t_2) : ?A} \text{(con)} \\ \frac{\models ?\Gamma, s : A \quad i \text{ fresh}}{\models \square_i(?\Gamma), !_i s : !A} \text{(prom)} & \frac{\models \Gamma_1, t : A \quad \models \Gamma_2, s : A^\perp}{\models \Gamma_1, \Gamma_2, \mathbf{cut}_A(t, s) : \perp} \text{(cut)} \end{array}$$

where in each derivation rules with two premises, $\frac{\models \Gamma_1 \quad \models \Gamma_2}{\models \Gamma}$, we implicitly assume that Γ_1 and Γ_2 have disjoint sets of basic variables and indexes, Γ denotes any possible set of proof terms, $? \Gamma$ denotes a set of proof terms in the form $t_1 : ?A_1, \dots, t_n : ?A_n, s_1 : \perp, \dots, s_m : \perp$, for $m, n \geq 0$.

Given $? \Gamma = t_1 : ?A_1, \dots, t_n : ?A_n, s_1 : \perp, \dots, s_m : \perp$, the expression $\square_i(? \Gamma)$ denotes

$$\square_i(t_1) : ?A_1, \dots, \square_i(t_n) : ?A_n, \square_i(s_1) : \perp, \dots, \square_i(s_m) : \perp.$$

Note that in the compact notation for proof terms, in order to be able to associate a MELL formula to each sub-proof term, we mark each application of cut with a MELL formula. Moreover, it is necessary to choose a type variable α for each axiom rule, and an index variable for each promotion rule. These choices are completely arbitrary, and therefore we consider equivalent proof-nets that differ only by variable relabelling.

It is not difficult to check that the above rules are in direct correspondence with MELL rules.

4.1 Lambda nets

In this paper, we are interested in λ -nets, that is in proof nets representing λ -terms. These particular proof nets are characterised by introducing polarity on formulas and by restricting the occurrences of \wp and \otimes according to polarities. To model the pure untyped λ -calculus,

one needs to define a structure isomorphic to its own function space, $D = D \rightarrow D$; in the linear logic encoding this corresponds to consider a proposition O such that $O = !O \multimap O = ?(O^\perp) \wp O$. As a consequence, we consider proof structures having as conclusion just four kinds of propositions, $O, !O, I$ and $?I$, related by $I = O^\perp$, and satisfying the equivalence induced by $O = !O \multimap O$. This in particular implies $O = ?I \wp O$ and $I = !O \otimes I$.

The grammar for proof terms is the following:

$$\begin{aligned}
t : C ::= & \alpha : O \mid \bar{\alpha} : I \\
& \mid (s : ?I) \wp (t : O) : O \mid (t : !O) \otimes (s : I) : I \\
& \mid !_i(t : O) : !O \mid \mathbf{d}(s : I) : ?I \mid \mathbf{c}((s_1 : ?I), (s_2 : ?I)) : ?I \\
& \mid \square_i(t : ?I) : ?I \mid \square_i(s : \perp) : \perp \\
& \mid \mathbf{cut}(s : I, t : O) : \perp \mid \mathbf{cut}_!(s : ?I, t : !O) : \perp \\
& \mid \mathbf{w} : ?I
\end{aligned}$$

In the following, we are going to consider two classes of λ -nets, the standard ones and a modified version. Since we are dealing with a type system where only strongly normalisable terms are typable, and principal types correspond to λ -nets in normal form, we need to define a cut-elimination that is perpetual, *i.e.* it converges on a λ -net if the standard cut-elimination procedure converges for every possible choice of order in which cuts are reduced. This can be achieved by modifying the way weakening is defined. While in a standard cut-elimination procedure boxes containing cuts can be removed, in our approach boxes are only partially eliminated and the cuts that they contain are left pending, so the cut-elimination procedure needs to explicitly eliminate all the cuts which are present in the λ -net.

In the following, we present two rules for weakening, the standard and the modified one: standard λ -nets are built using standard weakening, *i.e.* rule (s-weak), while modified λ -nets are constructed using modified weakening, *i.e.* rule (m-weak). Note that in a modified λ -net a type variable, α or $\bar{\alpha}$, can appear just once. In the graphical representation of λ -nets, this possibility corresponds to admitting axioms with a single component.

λ -structures are then defined as sets of proof terms containing a single term with type O or $!O$, with all other terms having types $I, ?I$ or \perp .

The set of rules defining λ -nets are the following:

$$\begin{aligned}
\frac{}{\models \bar{\alpha} : I, \alpha : O} \text{(ax)} \quad & \frac{}{\models \Gamma, s : ?I, t : O} \frac{}{\models \Gamma, s \wp t : O} \text{(par)} \quad \frac{}{\models \Gamma_1, s : I} \frac{}{\models \Gamma_2, t : !O} \frac{}{\models \Gamma_1, \Gamma_2, t \otimes s : I} \text{(tens)} \\
\frac{}{\models \Gamma, s : I} \frac{}{\models \Gamma, \mathbf{d} s : ?I} \text{(der)} \quad & \frac{}{\models \Gamma, s_1 : ?I, s_2 : ?I} \frac{}{\models \Gamma, \mathbf{c}(s_1, s_2) : ?I} \text{(con)} \quad \frac{}{\models ?\Gamma, t : O} \frac{i \text{ fresh}}{\models \square_i(?\Gamma), !_i t : !O} \text{(prom)} \\
\frac{}{\models \Gamma_1, s : I} \frac{}{\models \Gamma_2, t : O} \frac{}{\models \Gamma_1, \Gamma_2, \mathbf{cut}(s, t) : \perp} \text{(cut)} \quad & \frac{}{\models \Gamma_1, s : ?I} \frac{}{\models \Gamma_2, t : !O} \frac{}{\models \Gamma_1, \Gamma_2, \mathbf{cut}_!(s, t) : \perp} \text{(cut}_! \text{)} \\
\frac{}{\models \Gamma} \frac{}{\models \Gamma, \mathbf{w} : ?I} \text{(s-weak)} \quad & \frac{}{\models \Gamma_1, t_1 : O} \frac{}{\models \Gamma_2, t_2 : O} \frac{}{\models \Gamma_1, \Gamma_2, t_1 : O} \text{(m-weak)}
\end{aligned}$$

where in each derivation rules with two premises, $\frac{}{\models \Gamma} \frac{}{\models \Gamma_1} \frac{}{\models \Gamma_2}$, we implicitly assume that Γ_1 and Γ_2 have disjoint sets of basic variables and indexes.

In this setting, the cut-elimination steps are defined by the following rules:

$$\begin{aligned}
\models \Gamma, \mathbf{cut}(s, \alpha) : \perp & \rightarrow_{ces} \models \Gamma[s/\bar{\alpha}] \\
\models \Gamma, \mathbf{cut}(\bar{\alpha}, t) : \perp & \rightarrow_{ces} \models \Gamma[t/\alpha] \\
\models \Gamma, \mathbf{cut}(t_1 \otimes s_1, s_2 \wp t_2) : \perp & \rightarrow_{ces} \models \Gamma, \mathbf{cut}(s_1, t_2) : \perp, \mathbf{cut}_!(s_2, t_1) : \perp
\end{aligned}$$

$$\begin{aligned}
& \models \Gamma, \mathbf{cut}_!(\mathbf{c}(s_1, s_2), !_i t) : \perp \rightarrow_{ces} \vdash \Gamma(\mathit{Dup}(i)), \mathbf{cut}_!(s_1, \mathit{VerL}(!_i t)) : \perp, \mathbf{cut}_!(s_2, \mathit{VerR}(!_i t)) : \perp \\
& \models \Gamma, \mathbf{cut}_!(\Box_j s, !_i t) : \perp \rightarrow_{ces} \models \Gamma[j, i / i], \Box_j(\mathbf{cut}_!(s, !_i t)) : \perp \\
& \models \Gamma, \mathbf{cut}_!(\mathbf{d} s, !_i t) : \perp \rightarrow_{ces} \models \Gamma[\epsilon / i], \mathbf{cut}(s, t) : \perp \\
& \models \Gamma, \mathbf{cut}_!(\mathbf{w}, !_i t) : \perp \rightarrow_{ces} \models \Gamma(\mathit{Del}(i))
\end{aligned}$$

where with $\Gamma[t/\alpha]$, $\Gamma[j, i / i]$, $\Gamma[\epsilon / i]$, $\Gamma(\mathit{Dup}(i))$ we denote respectively the environment Γ where a suitable operation has been performed: namely, in $\Gamma[t/\alpha]$, the substitution of a variable α by the term t ; in $\Gamma[j, i / i]$, the substitution of each \Box_i by the sequence $\Box_j \Box_i$; in $\Gamma[\epsilon / i]$, the cancellation of each \Box_i ; in $\Gamma(\mathit{Dup}(i))$, the duplication of each subterm in the form $\Box_i t$, implemented similarly to the MGU algorithm; in $\Gamma(\mathit{Del}(i))$, the substitution of each subterm in the form $\Box_i t$ by \mathbf{w} .

Notice that the first two rules for cut-elimination correspond to the standard rule for the axiom case, the 3rd rule corresponds to the standard rule for par-tensor, the 4th rule corresponds to the standard rule for the promotion-promotion case, where one box is inserted into the other, the 5th rule corresponds to rule for dereliction-promotion where a box is eliminated, and the 6th rule corresponds to the contraction-promotion case where a box is duplicated. The last rule is the weakening-promotion case, and it will be applied only to λ -nets obtained through the standard translation of λ -terms.

Notice moreover that there is an, at the moment informal, one to one correspondence between the MGU-rules and the first 6 rules of cut-elimination. This correspondence will be formalised later.

4.2 From λ -terms to λ -nets

In this section we present two translations from λ -terms in the two classes of λ -nets: a translation \mathcal{S} to standard λ -nets, mimicking the standard translation, [20], and a modified translation, \mathcal{L} , in the modified λ -nets. We will state some properties explaining how the two translations are connected; this will allow us to derive properties of \mathcal{L} from already known properties of \mathcal{S} .

► **Definition 18.** *We associate to a λ -term M two λ -structures, $\mathcal{L}(M)$ and $\mathcal{S}(M)$, where the types $?I$ appearing outermost are marked with the free variables of M . The λ -net $\mathcal{L}(M)$ is inductively defined as follows:*

$$\begin{aligned}
\mathcal{L}(x) &= \models \mathbf{d} \bar{\alpha} : ?I_x, \alpha : O \\
\mathcal{L}(M) &= \models \Gamma_1, t : O \quad \mathcal{L}(N) = \models \Gamma_2, s : O \quad \alpha \text{ fresh} \\
& \quad \text{Var}(\Gamma_1, t : O) \cap \text{Var}(\Gamma_2, s : O) = \emptyset \\
\hline
\mathcal{L}(MN) &= \models \mathbf{c}(\Gamma_1, \Box_i \Gamma_2), \mathbf{cut}(!_i s \otimes \bar{\alpha}), t : \perp, \alpha : O
\end{aligned}$$

$$\begin{aligned}
\frac{\mathcal{L}(M) = \models \Gamma, s : ?I_x, t : O}{\mathcal{L}(\lambda x.M) = \models \Gamma, s \wp t : O} \quad & \frac{\mathcal{L}(M) = \models \Gamma, t : O \quad x, \alpha \text{ fresh}}{\mathcal{L}(\lambda x.M) = \models \Gamma, (\mathbf{d} \bar{\alpha}) \wp t : O}
\end{aligned}$$

where $\mathbf{c}(\Gamma, \Gamma') = \{\mathbf{c}(s, t) : ?I_x \mid (s \in ?I_x) \in \Gamma \wedge (t : ?I_x) \in \Gamma'\} \cup \{s : ?I_x \mid (s \in ?I_x) \in \Gamma \wedge \nexists t. (t : ?I_x) \in \Gamma'\} \cup \{s : ?I_x \mid (s \in ?I_x) \in \Gamma' \wedge \nexists t. (t : ?I_x) \in \Gamma\}$.

The standard λ -net $\mathcal{S}(M)$ is inductively defined by the same rules as above, just the last rule is reformulated as:

$$\begin{aligned}
\mathcal{S}(M) &= \models \Gamma, t : O \quad x \text{ fresh} \\
\mathcal{S}(\lambda x.M) &= \models \Gamma, (\mathbf{w} \wp t) : O
\end{aligned}$$

Notice that in the above definition one needs to choose the type and index variables appearing in $\mathcal{L}(M)$, however, all choices generate equivalent λ -nets.

► **Lemma 19.** *The function \mathcal{L} associates to each λ -term M a λ -net in the form $\mathcal{L}(M) = \models_{s_1 : ?I_{x_1}, \dots, s_n : ?I_{x_n}, t_1 : \perp, \dots, t_k : \perp, t : O}$, where $\{x_1, \dots, x_n\} = FV(M)$, k gives the number of cuts, which corresponds to the number of applications in the λ -term. A similar result holds for \mathcal{S} .*

Proof. By induction on the structure of M . One can easily check that the λ -structure $\mathcal{L}(M)$ is a λ -net, using the inductive hypothesis and a suitable set of derivation rules. The correspondence is the following:

- for a variable one uses (ax) and (der) rules;
- for a λ -abstraction binding a variable appearing in the body of the term, one uses the (par) rule;
- for a λ -abstraction binding a variable not appearing in the body of the term, one uses (ax), (der), (m-weak), and (par) rules;
- for application, one uses (prom), (ax), (tens), (cut), and (con) rules.

An analogous proof can be formulated for \mathcal{S} . ◀

Notice that, given a λ -term M in normal form, the translations $\mathcal{L}(M)$ and $\mathcal{S}(M)$ are λ -nets *not* in normal form, however they can be reduced to normal form by a number of cut-elimination steps all of the shape $\models \Gamma, \mathbf{cut}(s : I, \alpha : O) : \perp \rightarrow_{ces} \models \Gamma[s/\bar{\alpha}]$. Quite often proof nets that differ just by the application of the above cut-elimination steps are informally considered equivalent.

The translation \mathcal{L} differs from the standard translation \mathcal{S} of λ -calculus into λ -nets because of the weakening rule. This difference leads to a difference in the way cut-elimination is performed. The standard translation leads to cuts between exponentials and weakening constants, where the whole box containing the exponential is eliminated. The new translation leads to cuts between unpaired variables and exponentials, where not the whole box containing the exponential is eliminated, but just the exponential terms, while the remaining parts of the box are left unchanged, just the box boundary is removed. As a consequence, given a closed term M , the cut in $\mathcal{L}(M)$ can be eliminated if and only if M is strongly normalisable, and this is due to the fact that in cut-elimination no cut is deleted without being resolved.

In the following we present a formal proof of the above facts.

► **Definition 20.**

- (i) Let \approx be the least congruent order relation on proof terms and λ -nets s.t. $\mathbf{w} : ?I \approx \mathbf{d} \bar{\beta} : ?I$, for any type variable β .
- (ii) Let \lesssim be the least congruent order relation on proof terms and λ -nets such that:
 - $\mathbf{w} : ?I \lesssim \mathbf{d} \bar{\beta} : ?I$, $t : ?I \lesssim \mathbf{c}(t, t') : ?I$
 - $\models \Gamma \lesssim \models \Gamma, t : ?I$, $\models \Gamma \lesssim \models \Gamma, t : \perp$, and $\models \Gamma, t : A \lesssim \models \Gamma, t' : A$ whenever $t : A \lesssim t' : A$.

A lemma relating \approx and \lesssim is the following:

► **Lemma 21.** *If $\models \Gamma \approx \models \Gamma'$, $\models \Gamma \lesssim \models \Delta$ and Δ does not contain the constant \mathbf{w} , then there exists $\models \Gamma''$, which coincides, up-to renaming of type variables, with $\models \Gamma'$, and such that $\models \Gamma'' \lesssim \models \Delta$.*

Proof. Any instance of \mathbf{w} appearing in Γ must be replaced by a proof terms $t : ?I$ with $\mathbf{d} \bar{\beta} : ?I \lesssim t : ?I$ (for some type variable β) in Δ , while the same instance of \mathbf{w} , can remain the same or be replaced by a proof term $\mathbf{d} \bar{\alpha} : ?I$ in Γ' . The λ -net $\models \Gamma''$ is obtained from $\models \Gamma'$ by replacing any such α in Γ' by the corresponding β appearing in Δ . ◀

5:16 Principal Types as Lambda Nets

The relation \lesssim is preserved by the cut-eliminations steps, that is:

- **Lemma 22.** *For any pair of λ -nets $\models\Gamma$ and $\models\Delta$ such that $\models\Gamma \lesssim \models\Delta$, the following hold:*
- *for any Γ' such that $\models\Gamma \rightarrow_{ces} \models\Gamma'$, there exists Δ' such that $\models\Delta \rightarrow_{ces} \models\Delta'$ and $\models\Gamma' \lesssim \models\Delta'$.*
 - *for any Δ' such that $\models\Delta \rightarrow_{ces} \models\Delta'$, either $\models\Gamma \lesssim \models\Delta'$ or there exists Γ' such that $\models\Gamma \rightarrow_{ces} \models\Gamma'$ and $\models\Gamma' \lesssim \models\Delta'$.*

$$\begin{array}{ccc} \models\Gamma & \lesssim & \models\Delta \\ \downarrow_{ces} & & \downarrow_{ces} \\ \models\Gamma' & \lesssim & \models\Delta' \end{array}$$

Proof. Both items can be proved by case analysis on the kind of the cut-elimination step, and by observing that, when $\models\Gamma \lesssim \models\Delta$, any subterm in Γ appears, possibly in an extended form (i.e. $\mathbf{c}(t, t')$ in place of t) also in Δ , therefore any reduction in Γ can be replicated in Δ . ◀

The above lemma, whose proof is immediate, relates the λ -nets $\mathcal{S}(M)$ and $\mathcal{L}(M)$ for any term M .

- **Lemma 23.** *For any term M , $\mathcal{S}(M) \lesssim \mathcal{L}(M)$ and $\mathcal{S}(M) \lesssim \mathcal{L}(M)$.*

The following lemmas show that β -reduction on λ -terms can be simulated (even not faithfully) by cut-elimination.

- **Lemma 24.** *For any pair of λ -terms M and N , if $M \rightarrow_{\beta} N$, then there exists a λ -net $\models\Gamma$ such that $\mathcal{L}(M) \rightarrow_{ces} \models\Gamma$ and $\mathcal{L}(N) \lesssim \models\Gamma$.*

Proof. By the properties of λ -nets, see e.g. [19], we have that $\mathcal{S}(M) \rightarrow_{ces} \mathcal{S}(N)$, then by Lemmas 23 and 22, there exists $\models\Gamma$ such that $\mathcal{L}(M) \rightarrow_{ces} \models\Gamma$ and $\mathcal{S}(N) \lesssim \models\Gamma$. Applying Lemmas 21 and 23, we derive the thesis. ◀

Notice that if M does not contain any affine abstraction, $\mathcal{L}(M)$ coincides with $\mathcal{S}(M)$, and so if M β -reduces to N , also $\mathcal{L}(M)$ reduces to $\mathcal{L}(N)$.

A subject reduction property for λ -nets in normal form is the following:

- **Lemma 25.** *For any pair of λ -terms M and N , if $M \rightarrow_{\beta} N$, and $\mathcal{L}(M)$ normalises to a cut-free λ -net $\models\Gamma$, then also $\mathcal{L}(N)$ normalises to a cut-free λ -net $\models\Gamma'$, and $\models\Gamma' \lesssim \models\Gamma$.*

Proof. By Lemma 24 there exists $\models\Delta$ such that $\mathcal{L}(M) \rightarrow_{ces} \models\Delta$ and $\mathcal{L}(N) \lesssim \models\Delta$. Since cut-elimination satisfies the Church-Rosser property, it is possible from $\models\Delta$ to reduce to $\models\Gamma$. By multiple applications of Lemma 22, $\mathcal{L}(N)$ reduces to a λ -net $\models\Gamma'$ satisfying $\models\Gamma' \lesssim \models\Gamma$. Hence $\models\Gamma'$ does not contain any cut, that is, is in normal form.

$$\begin{array}{ccc} \mathcal{L}(M) - ces \rightarrow \models\Delta & - ces^* \rightarrow & \models\Gamma \\ \vdots & & \vdots \\ \wr\vee & & \wr\vee \\ \vdots & & \vdots \\ \mathcal{L}(N) - ces^* \rightarrow & \models\Gamma' & \end{array}$$

For λ -nets not containing the constant \mathbf{w} all reduction strategies are equivalent from the termination point of view. ◀

► **Lemma 26.** *For any λ -net $\models\Gamma$ not containing the constant \mathbf{w} , either the cuts cannot be completely eliminated from $\models\Gamma$ or any possible sequence of cut-elimination steps leads to a normal form.*

Proof. Let corecursively define a cut *divergent* if, when eliminated by the associated elimination rule, it generates new cuts and one of these is divergent. Notice that no rule can create a new divergent cut, unless a divergent cut already exists in the λ -net. If a cut-elimination sequence diverges on a λ -net $\models\Gamma$, then $\models\Gamma$ must contain a divergent cut. If the constant \mathbf{w} is not present in a λ -net $\models\Gamma$, no rule eliminating boxes can be applied, so no cut can be eliminated as a consequence of the elimination of a box. It follows that, if $\models\Gamma$ can diverge, it contains a divergent cut, then divergent cuts cannot be completely eliminated from $\models\Gamma$, no matter in what order the reduction steps are applied. ◀

The previous proposition implies that, for any λ -term M , strong normalisation and weak normalisation on $\mathcal{L}(M)$ coincide.

► **Proposition 27.** *For any λ -term M , the following conditions are all equivalent:*

- (i) $\mathcal{L}(M)$ is weakly normalising.
- (ii) $\mathcal{L}(M)$ is strongly normalising.
- (iii) $\mathcal{S}(M)$ is strongly normalising.
- (iv) M is strongly normalising.

Proof. By Lemma 26, conditions (i) and (ii) are equivalent.

The implication (ii) \Rightarrow (iii) is proved using Lemmas 23 and 22. In fact, by these lemmas, any reduction starting from $\mathcal{S}(M)$ can be lifted to a reduction starting from $\mathcal{L}(M)$. Since the lifted reduction terminates, also the original reduction terminates.

To prove (iii) \Rightarrow (i), one starts with $\mathcal{S}(M)$ and applies a reduction strategy where no cut rule for \mathbf{w} is used if the eliminated box contains a cut. In this case one eliminates first the cut inside the box. Since $\mathcal{S}(M)$ is strongly normalising, in this way we obtain a terminating chain C of reductions. Using the construction in Lemma 22, the chain C can be lifted to a sequence of reductions starting from $\mathcal{L}(M)$. Since no box containing a cut is eliminated, the lifted chain does not contain extra cuts, hence this leads to a cut-free λ -net, that is it terminates.

The equivalence between (iii) and (iv) is a standard result in the theory of proof nets, [20]. ◀

4.3 From principal types to λ -nets

We define functions $\mathcal{P}_p, \mathcal{P}_n$ transforming positive and negative principal types in proof terms representing parts of proof nets:

- $\mathcal{P}_p(\alpha) = \alpha : O$
- $\mathcal{P}_p(\hat{\sigma} \multimap \tau) = (\mathcal{P}_n(\hat{\sigma}) \wp \mathcal{P}_p(\tau)) : O$
- $\mathcal{P}_p(\Box_i! \tau) = (!_i \mathcal{P}_p(\tau)) : !O$
- $\mathcal{P}_n(\bar{\alpha}) = \bar{\alpha} : I$
- $\mathcal{P}_n(\hat{\tau} \multimap \sigma) = (\mathcal{P}_p(\hat{\tau}) \otimes \mathcal{P}_n(\sigma)) : I$
- $\mathcal{P}_n(!\sigma) = \mathbf{d}(\mathcal{P}_n(\sigma)) : ?I$
- $\mathcal{P}_n(\Box_i \hat{\sigma}) = \Box_i(\mathcal{P}_n(\hat{\sigma})) : ?I$
- $\mathcal{P}_n(\hat{\sigma}_1 \wedge \hat{\sigma}_2) = \mathbf{c}(\mathcal{P}_n(\hat{\sigma}_1), \mathcal{P}_n(\hat{\sigma}_2)) : ?I$

The translations above induce a translation \mathcal{P} from judgements to cut-free λ -nets, where the types in the proof net are indexed by free variables:

$$\mathcal{P}(x_1 : \hat{\sigma}_1, \dots, x_n : \hat{\sigma}_n \Vdash M : \tau) = \models \mathcal{P}_n(\hat{\sigma}_1)_{x_1}, \dots, \mathcal{P}_n(\hat{\sigma}_n)_{x_n}, \mathcal{P}_p(\tau).$$

First of all, we have the following lemma (whose proof is immediate):

► **Lemma 28.** *The function \mathcal{P}_p defines an isomorphism between positive simple-types and proof terms having type in the form $t : O$, and between positive and-types and proof terms in the form $t : !O$; the function \mathcal{P}_n defines an isomorphism between negative simple-types and proof terms having type in the form $t : I$, and between negative and-types and proof terms in the form $t : ?I$.*

The main result of this section consists in showing that a judgement $\Delta \Vdash M : \tau$ is derivable if and only if $\mathcal{L}(M)$ reduces to the cut-free λ -net $\mathcal{P}(\Delta \Vdash M : \tau)$. In order to show this, we need first to establish a correspondence between performing cut-elimination and evaluating a MGU and applying it to a judgement. This correspondence is expressed by the following lemma:

► **Lemma 29.** *For any negative type ν and positive type μ , either both simple or both and-types, and for any judgement $\Delta \Vdash M : \tau$,*

- *MGU(ν, μ) exists iff the λ -net $\models \mathbf{cut}(\mathcal{P}_n(\nu), \mathcal{P}_p(\mu))$ reduces to a cut-free λ -net.*
- *If MGU(ν, μ) exists, then:*
 - $\mathcal{P}(\text{MGU}(\nu, \mu)(\Delta \Vdash M : \tau))$ *is equal to the normal form of* $\models \Gamma, \mathbf{cut}(\mathcal{P}_n(\nu), \mathcal{P}_p(\mu)) : \perp$, *where* $\Gamma = \mathcal{P}(\Delta \Vdash M : \tau)$.

Proof. The proof is by induction on the derivation $\text{MGU}(\nu, \mu) = U$, for some unification U . Informally, the correspondence can be explained in the following way. When in a proof net $\models \Gamma, \mathbf{cut}(\mathcal{P}_n(\nu), \mathcal{P}_p(\mu)) : \perp$ the cut $\mathbf{cut}(\mathcal{P}_n(\nu), \mathcal{P}_p(\mu)) : \perp$ is eliminated, the remaining part of the proof is affected in several ways, *i.e.* some variables are substituted, some boxes are duplicated, the boundaries of some boxes are deleted, some boxes are inserted in other boxes. All these actions coincide with the actions generated by the $\text{MGU}(\nu, \mu)$. ◀

Comparing the MGU definition to cut-elimination, one observes that, while the MGU is essentially deterministic, *i.e.* there is always a single rule to apply (up-to the axioms, in the case of two variable types), the cut-elimination procedure is non-deterministic, *i.e.* one can choose the order in which cuts are eliminated. So the MGU evaluation is in direct correspondence with one particular strategy of cut-elimination. However, in Proposition 27 we have shown that all strategies on $\mathcal{L}(M)$ are equivalent from the normalisation point of view.

We are now in the position to state the following:

► **Proposition 30.** *For any Δ, M, τ , the principal type judgement $\Delta \Vdash M : \tau$ is derivable iff the λ -net $\mathcal{L}(M)$ reduces, by cut-elimination, to the cut-free λ -net $\mathcal{P}(\Delta \Vdash M : \tau)$.*

Proof. The proof is by induction on the structure of M , all cases but application are immediate. For the application case, $M = M_1N$, one distinguishes two subcases. The first is when the normal form of $\mathcal{L}(M_1)$ has the shape $\models \Gamma, \alpha : O$. In this case, one observes that the cut introduced by the main term application can be eliminated in one step, by a single substitution, and the normal form of $\mathcal{L}(M_1N)$ coincides with the λ -net obtained by \mathcal{P} -translation of a judgement derived by the (appNorm)-rule. When this case does not apply, *i.e.*, $\mathcal{L}(M)$ is not in the form $\models \Gamma, \alpha : O$, the thesis follows from Lemma 29 and from the fact that $\mathcal{L}(M)$ is weakly normalising iff it is strongly normalising, by Proposition 27. ◀

5 Applications

In this section we aim at showing how the various correspondence results established in the previous sections (between the two typing systems, λ -nets and the principal typing system, the two translations of λ -terms into λ -nets) can be exploited to prove properties of the ground typing system of Definition 2.

The properties that we are going to analyse are strong normalisation, subject reduction, and inhabitation. Such properties are already known, and they have been extensively studied in the literature, but here we present alternative proofs based on the above correspondence results.

All proofs in this section follow this basic pattern:

- given a property of λ -terms, we associate a property of λ -nets via the standard interpretation;
- via the correspondence between standard and modified interpretations, we associate a property of λ -nets through the modified interpretation;
- via the correspondence between modified interpretation and the principal typing system, we associate a property on principal types;
- finally, via the correspondence between the principal typing system and the ground system, we derive a property of the ground typing assignment system.

5.1 Typability and strong normalisation

Idempotent or non-idempotent intersection types are quite often used to characterise (strongly) normalisable terms, [2, 22, 3, 6]. By extending the chain of equivalences of Proposition 27, we can give an alternative proof of a classical result, using the correspondence between principal types and lambda-nets:

► **Proposition 31.** *For all $M \in \Lambda^0$, the following conditions are equivalent:*

- (i) M is strongly normalising.
- (ii) M has a principal type.
- (iii) M has a ground type.

Proof. By Proposition 27, M is strongly normalising iff $\mathcal{L}(M)$ is normalisable, by Proposition 30 this is equivalent to (ii), which in turn is equivalent to (iii) by Theorem 16. ◀

5.2 Subject reduction

Subject reduction is a sort of minimal requirement for any typing system; again one can use the correspondence between principal types and lambda-nets to derive this standard result. In particular, subject reduction holds up-to a suitable \lesssim -relation on types. Notice that, however, subject conversion fails, as it is always the case for typing systems characterising strongly normalisable terms.

► **Definition 32.** *Let \lesssim be the least precongruence on (principal) types extending the relation $\hat{\sigma}_1 \lesssim \hat{\sigma}_1 \wedge \hat{\sigma}_2$.*

Precongruences on types and on λ -nets are related by:

► **Lemma 33.** *For all positive types μ_1, μ_2 , $\mu_1 \lesssim \mu_2$ iff $\mathcal{P}_p(\mu_1) \lesssim \mathcal{P}_p(\mu_2)$.*

► **Theorem 34 (Subject Reduction).** *For all $M \in \Lambda^0$,*

$$\vdash M : \tau \ \& \ M \rightarrow_{\beta} N \implies \exists \tau'. \vdash N : \tau' \ \& \ \tau' \lesssim \tau .$$

Proof. By Theorem 16, M has a principal type, hence, by Proposition 31, M is strongly normalisable. Therefore also N is strongly normalisable, and by Proposition 31 it has a principal type. By Proposition 30, both $\mathcal{L}(M)$ and $\mathcal{L}(N)$ reduce to normal forms. Since M and N are closed, such λ -nets are composed by a single proof term, $\models t_1 : O$ and $\models t_2 : O$. Moreover, let τ_1 and τ_2 be the type schemata such that $\mathcal{P}_p(\tau_1) = t_1$ and $\mathcal{P}_p(\tau_2) = t_2$ (they exist since \mathcal{P}_p is an isomorphism). By Lemmas 24 and 22, $\models t_2 : O \lesssim \models t_1 : O$, and therefore, by Lemma 33, $\tau_2 \lesssim \tau_1$. By Proposition 30 $\Vdash M : \tau_1$ and $\Vdash N : \tau_2$. By Theorem 16(i), there exists a substitution-replication T such that $\tau = T(\tau_1)$. Since $\tau_2 \lesssim \tau_1$, it follows that $T(\tau_2)$ is a ground type, hence $\Vdash N : T(\tau_2)$, and $T(\tau_2) \lesssim \tau$. ◀

5.3 Inhabitation

Here we show that the inhabitation problem for principal types can be reduced to the problem of correctness of λ -structures, and therefore it is decidable. Then we reflect this result to the ground typing system, obtaining an alternative proof of the decidability result proved in [5]. Here the hypothesis of non-idempotency of the \wedge -operator is essential.

► **Proposition 35.** *For any list of negative and-types $\hat{\sigma}_1, \dots, \hat{\sigma}_n$, and for any positive type τ , there exists a λ -term M with free variables x_1, \dots, x_n such that: $x_1 : \hat{\sigma}_1, \dots, x_n : \hat{\sigma}_n \Vdash M : \tau$ if and only if the λ -structure $\models \mathcal{P}_n(\hat{\sigma}_1), \dots, \mathcal{P}_n(\hat{\sigma}_n), \mathcal{P}_p(\tau)$ is a λ -net.*

Proof.

(\Rightarrow) If M exists then, by Proposition 30, $\models \mathcal{P}_n(\hat{\sigma}_1), \dots, \mathcal{P}_n(\hat{\sigma}_n), \mathcal{P}_p(\tau)$ is the normal form of $\mathcal{L}(M)$, and therefore a λ -net.

(\Leftarrow) By structural induction on the derivation of λ -nets, we show that, for any λ -net in the form $\models s_1 : I, \dots, s_m : I, t_1 : ?I, \dots, t_n : ?I, t : O$, there exists a λ -term M with free variables $x_1, \dots, x_m, y_1, \dots, y_n$ such that each variable x_i appears once in M , and the normal form of $\mathcal{L}(M)$ is the λ -net $\models \mathbf{d} s_1 : ?I_{x_1}, \dots, \mathbf{d} s_m : ?I_{x_m}, t_1 : ?I_{y_1}, \dots, t_n : ?I_{y_n}, t : O$ (up-to commutativity and associativity of the contraction operator, **c**). Then the thesis follows from Proposition 30. In proving the above fact, many of the inductive cases are quite straightforward, here we consider the most difficult ones, namely (tens), (con), and (m-weak). In the following, let the metavariable Γ denote a sequence of proof terms in the form $t_1 : ?I, \dots, t_n : ?I$, while Δ denotes a sequence of proof terms in the form $s_1 : I, \dots, s_m : I$, in this case $\Delta?$ denotes the sequence of proof terms $\mathbf{d} s_1 : ?I, \dots, \mathbf{d} s_m : ?I$.

(tens) Let us suppose that $\models \Delta_1, \Gamma_1, \Gamma_2, t_2 \otimes s : I, t_1 : O$ is derived from $\models \Delta_1, \Gamma_1, s : I, t_1 : O$ and $\models \Gamma_2, t_2 : !O$. Notice that $\models \Gamma_2, t_2 : !O$ can only be derived through the (prom) rule. By inductive hypothesis there exist two terms $M[x]$ and N , such that $M[x]$ contains a variable x once and the normal form of $\mathcal{L}(M[x])$ has shape $\models \Delta_1?, \mathbf{d} s : ?I_x, \Gamma_1, t_1 : O$, while the normal form of $\mathcal{L}(N)$ has shape $\models \Gamma_2, t_2 : !O$. Quite obviously, the free variables in $M[x]$ and N can be chosen in such a way that no free variable is in common between the two terms. Now one can prove by structural induction on $M[x]$ that the normal form of $\mathcal{L}(M[x]N)$ has shape $\models \Delta_1?, \Gamma_1, \Gamma_2, \mathbf{d}(t_2 \otimes s) : ?I, t_1 : O$, from which the thesis follows.

(con) Let us suppose that $\models \Delta, \Gamma, \mathbf{c}(s_1, s_2) : ?I, t : O$ is derived from $\models \Delta, \Gamma, s_1 : ?I, s_2 : ?I, t : O$, by inductive hypothesis there exists a term $M[y_1, y_2]$ such that $\mathcal{L}(M[y_1, y_2])$ has shape $\models \Delta?, \Gamma, s_1 : ?I_{y_1}, s_2 : ?I_{y_2}, t : O$. Now one can prove, by structural induction on $M[y_1, y_2]$, that the normal form of $\mathcal{L}(M[y, y])$ has the shape (up-to commutativity and associativity of contraction) $\models \Delta, \Gamma, \mathbf{c}(s_1, s_2) : ?I_y, t : O$, from which the thesis follows.

(m-weak) Let us suppose that $\models \Delta, \Gamma, ?\Gamma', t : O$ is derived from $\models \Delta, \Gamma, t : O$ and $\models ?\Gamma', t' : O$. By inductive hypothesis, there exist terms M and N such that the normal forms of $\mathcal{L}(M)$ and $\mathcal{L}(N)$ have shapes $\models \Delta?, \Gamma, t : O$ and $\models ?\Gamma', t' : O$. Let x be a variable fresh in M , then it is easy to check that the normal form of $\mathcal{L}((\lambda x.M)N)$ has shape $\models \Delta?, \Gamma, ?\Gamma', t : O$. ◀

► **Proposition 36.** *The inhabitation problem for principal types is decidable.*

Proof. By Proposition 35, a type schema τ is inhabited iff the λ -structure $\models_{\mathcal{P}_p}(\tau)$ is a λ -net. Notice that, since we consider a modified version of the weakening rule, one cannot use the standard correctness criterion for λ -structures. However, by a simple induction on the complexity of λ -structures, it is easy to prove that the correctness problem is decidable also in the case of modified weakening. ◀

The previous result can then be transferred to the ground type system, obtaining an alternative proof of the decidability result of [5].

► **Proposition 37.** *The inhabitation problem for ground types is decidable.*

Proof. By Theorem 16, a ground type τ is inhabited by a term M if and only if there is a type schemata τ' inhabited by M , and such that τ is an instance of τ' . Now a ground type τ can be the instance of just a finite set of well-formed positive type schemata, $\{\tau_i \mid 0 < i \leq n\}$, that one can construct starting from τ . So the inhabitation of τ can be reduced to the inhabitation of the finite set of type schemata $\{\tau_i \mid 0 < i \leq n\}$, which is decidable. ◀

Notice that, in the proof of the above proposition, the hypothesis of non-idempotency of \wedge is essential. Namely, if \wedge is idempotent, it is not true that a ground type can be an instance of just a *finite* set of type schemata. In fact, it is known that the inhabitation problem is undecidable for ground types when \wedge is idempotent [27].

5.4 Normalisation algorithm for λ -terms

As a further development of the theory in the present paper, it is possible to use the type system for principal types to build a normalisation algorithm for λ -terms, that in turn can be seen as a normalisation algorithm based on proof nets, but working on terms and not on graphs. The code of the Haskell implementation of the above algorithm is available as supplementary material of the present submission in [14]. It is quite simple but not particularly efficient. Mainly, we wrote it as a first check of correctness for the formally given rules.

6 Final Remarks

We have illustrated the analogy of “principal types as λ -nets”, by focusing on a specific type assignment system for typing strongly normalising λ -terms. The correspondence allowed us to relate different concepts and to derive properties of the type assignment system from known results on λ -nets.

In our type assignment system, the \wedge operator is taken to be non-idempotent, however, all the results in the present paper, apart from inhabitation (Proposition 37), hold also in the case of idempotency of \wedge .

In this paper, we started from a type assignment system, and we built a modified version of λ -nets that naturally correspond to the principal types of our type assignment system. However, it is also possible to move the other way round, that is, start from a given notion of λ -net, define a corresponding principal type assignment system and from this build a corresponding ground type assignment system. If the construction is carried out correctly, the ground assignment system will automatically satisfy properties of subject reduction, strong or weak normalisation, etc. As future work, we plan to pursue this investigation by starting from standard λ -nets or from different encodings of the λ -calculus in proof nets, in particular

the one associated with the call-by-value reduction strategy, see [19], Section 4.2.1. The corresponding type assignment systems should characterise weakly (or strongly) normalising λ -terms w.r.t. different reduction strategies.

References

- 1 Beniamino Accattoli. Proof nets and the linear substitution calculus. In *International Colloquium on Theoretical Aspects of Computing*, volume 11187 of *LNCS*, pages 37–61. Springer, 2018. doi:10.1007/978-3-030-02508-3_3.
- 2 Henk Barendregt, Mario Coppo, and Mariangiola Dezani-Ciancaglini. A filter lambda model and the completeness of type assignment. *J. Symb. Log.*, 48(4):931–940, 1983. doi:10.2307/2273659.
- 3 Alexis Bernadet and Stéphane Lengrand. Non-idempotent intersection types and strong normalisation. *Log. Methods Comput. Sci.*, 9(4), 2013. doi:10.2168/LMCS-9(4:3)2013.
- 4 Gérard Boudol, Pierre-Louis Curien, and Carolina Lavatelli. A semantics for lambda calculi with resources. *Math. Struct. Comput. Sci.*, 9(4):437–482, 1999. URL: <http://journals.cambridge.org/action/displayAbstract?aid=44845>.
- 5 Antonio Bucciarelli, Delia Kesner, and Simona Ronchi Della Rocca. Inhabitation for non-idempotent intersection types. *Logical Methods in Computer Science*, 14(3), 2018. doi:10.23638/LMCS-14(3:7)2018.
- 6 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Strong normalization through intersection types and memory. In M. Benevides and R. Thiemann, editors, *Proceedings of the Tenth Workshop on Logical and Semantic Frameworks, with Applications (LSFA 2015)*, volume 23, pages 75–91. ENTCS, 2016. doi:10.1016/j.entcs.2016.06.006.
- 7 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017. doi:10.1093/jigpal/jzx018.
- 8 S. Carlier and J. B. Wells. Type inference with expansion variables and intersection types in system E and an exact correspondence with beta-reduction. In Eugenio Moggi and David Scott Warren, editors, *Proceedings of the 6th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 24-26 August 2004, Verona, Italy*, pages 132–143. ACM, 2004. doi:10.1145/1013963.1013980.
- 9 S. Carlier and J. B. Wells. Expansion: the crucial mechanism for type inference with intersection types: A survey and explanation. *Electron. Notes Theor. Comput. Sci.*, 136:173–202, 2005. doi:10.1016/j.entcs.2005.03.026.
- 10 S. Carlier and J. B. Wells. The algebra of expansion. *Fundam. Informaticae*, 121(1-4):43–82, 2012. doi:10.3233/FI-2012-771.
- 11 Alberto Ciaffaglione, Furio Honsell, Marina Lenisa, and Ivan Scagnetto. The involutions-as-principal types/application-as-unification analogy. In G. Barthe, G. Sutcliffe, and M. Veanes, editors, *LPAR-22*, volume 57 of *EPiC Series in Computing*, pages 254–270. EasyChair, 2018. doi:10.29007/ntwg.
- 12 Mario Coppo, Mariangiola Dezani-Ciancaglini, and Betti Venneri. Principal type schemes and λ -calculus semantics. In *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 536–560. Academic Press, 1980.
- 13 Daniel de Carvalho. *Semantiques de la logique lineaire et temps de calcul*. PhD thesis, Université Aix-Marseille II, 2007. URL: <https://theses.fr/2007AIX22066>.
- 14 P. Di Gianantonio. A typing and normalisation algorithm for lambda terms, 2019. URL: <https://users.dimi.uniud.it/~pietro.digianantonio/papers/code/principalTAS.hs>.
- 15 Stephen Dolan and Alan Mycroft. Polymorphism, subtyping, and type inference in mlsb. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*, pages 60–72. ACM, 2017. doi:10.1145/3009837.3009882.

- 16 E. Duquesne and J. Van De Wiele. A new intrinsic characterization of the principal type schemes. Research Report RR-2416, INRIA, 1995. Projet PARA. URL: <https://hal.inria.fr/inria-00074259>.
- 17 Thomas Ehrhard. A new correctness criterion for MLL proof nets. In *Proceedings of the Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS)*, pages 1–10. ACM, 2014. doi:10.1145/2603088.2603125.
- 18 Maribel Fernández and Ian Mackie. A calculus for interaction nets. In G. Nadathur, editor, *Principles and Practice of Declarative Programming*, volume 1702 of *LNCS*, pages 170–187. Springer, 1999. doi:10.1007/10704567_10.
- 19 Stefano Guerrini. Correctness of multiplicative proof nets is linear. In *Fourteenth Annual IEEE Symposium on Logic in Computer Science*, pages 454–463. IEEE Computer Science Society, 1999. doi:10.1109/LICS.1999.782640.
- 20 Stefano Guerrini. Proof nets and the lambda-calculus. In Thomas Ehrhard, editor, *Linear Logic in Computer Science*, pages 316–65. Cambridge University Press, 2004. doi:10.1017/CB09780511550850.
- 21 A.J. Kfoury and J.B. Wells. Principality and type inference for intersection types using expansion variables. *Theoretical Computer Science*, 311(1):1–70, 2004. doi:10.1016/j.tcs.2003.10.032.
- 22 D. Leivant. Typing and computational properties of lambda expressions. *Theoretical Computer Science*, 44:51–68, 1986.
- 23 Peter Møller Neergaard and Harry G. Mairson. Types, potency, and idempotency: Why nonlinearity and amnesia make a type system work. In *Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming, ICFP '04*, pages 138–149. ACM, 2004. doi:10.1145/1016850.1016871.
- 24 Simona Ronchi Della Rocca. Principal type scheme and unification for intersection type discipline. *Theor. Comput. Sci.*, 59:181–209, 1988. doi:10.1016/0304-3975(88)90101-6.
- 25 Laurent Régnier. *Lambda-Calcul et Réseaux*. PhD thesis, Université Paris VII, 1992. URL: <https://theses.fr/1992PA077165>.
- 26 Emilie Sayag and Michel Mauny. Characterization of the principal type of normal forms in an intersection type system. In V. Chandru and V. Vinay, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 335–346, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. doi:10.1007/3-540-62034-6_61.
- 27 Pawel Urzyczyn. The emptiness problem for intersection types. In *Proceedings IEEE Symposium on Logic in Computer Science*, pages 300–309, 1994. doi:10.1109/LICS.1994.316059.
- 28 J. B. Wells. The essence of principal typings. In *Proceedings of the 29th International Colloquium on Automata, Languages and Programming, ICALP '02*, pages 913–925. Springer-Verlag, 2002. doi:10.1007/3-540-45465-9_78.

Internal Strict Propositions Using Point-Free Equations

István Donkó ✉

Eötvös Loránd University, Budapest, Hungary

Ambrus Kaposi ✉ 

Eötvös Loránd University, Budapest, Hungary

Abstract

The setoid model of Martin-Löf’s type theory bootstraps extensional features of type theory from intensional type theory equipped with a universe of definitionally proof irrelevant (strict) propositions. Extensional features include a Prop-valued identity type with a strong transport rule and function extensionality. We show that a setoid model supporting these features can be defined in intensional type theory without any of these features. The key component is a point-free notion of propositions. Our construction suggests that strict algebraic structures can be defined along the same lines in intensional type theory.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases Martin-Löf’s type theory, intensional type theory, function extensionality, setoid model, homotopy type theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.6

Supplementary Material *Software (Formalisation)*: <https://bitbucket.org/akaposi/prop> [12] archived at `swh:1:dir:6648713cc70e9c6fa8a71cccaa31c1d91cfbc418`

Funding *István Donkó*: Supported by the ÚNKP-21-3 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

Ambrus Kaposi: Supported by the Bolyai Fellowship of the Hungarian Academy of Sciences, Project no. BO/00659/19/3, and by the “Application Domain Specific Highly Reliable IT Solutions” project which has been implemented with the support provided from the National Research, Development and Innovation Fund of Hungary, financed under the Thematic Excellence Programme TKP2020-NKA-06 (National Challenges Subprogramme) funding scheme.

Acknowledgements Thanks to Christian Sattler for discussions on the topics of this paper. Many thanks to the anonymous reviewers for their comments and suggestions.

1 Introduction

The setoid model of type theory pioneered by Hofmann [15] supports the following extensional features that are missing from intensional type theory: function extensionality, propositional extensionality (univalence for propositions [4]) and quotient inductive-inductive types [18]. If the setoid model is defined in an intensional metatheory and all equations of the model (such as the β rule) hold definitionally, then it constitutes a *model construction* (also called syntactic translation): any model of intensional type theory can be turned into its “setoidified” variant which supports the extensional features, thus bootstrapping the extensional features from intensional type theory. Hofmann’s original model only justified some of the equations definitionally. Altenkirch showed that if the metatheory supports a sort TyP of definitionally proof irrelevant propositions in addition to the sort Ty of types, then there is a version of the setoid model where all equations are definitional [2]. After he presented this model construction at the Symposium on Logic in Computer Science in 1999 [2], Per Martin-Löf asked whether it is possible to remove the extra requirement of TyP . As far as we know, the question is still open.



© István Donkó and Ambrus Kaposi;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 6; pp. 6:1–6:21

Leibniz International Proceedings in Informatics



LIPIC Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

6:2 Internal Strict Propositions Using Point-Free Equations

In this setoid model a closed type is a setoid: a type together with an equivalence relation; a term is a function between the types which respects the relations. If the equivalence relation is proof relevant (Ty-valued), then terms have to additionally include components about respecting the reflexivity, symmetry and transitivity proofs, then when proving equalities of terms (such as the β law), one has to show that the corresponding new components are equal, which forces the introduction of new components, and so on. This problem is usually referred to as coherence problem, see [15, Section 5.3] for a discussion in the context of the setoid model, or [19] for a recent exposition of the general phenomenon. Altenkirch’s solution [2] was to make the relation TyP-valued instead of Ty-valued: in this case, terms automatically respect reflexivity proofs as there is only one proof of reflexivity, up to definitional equality. We could avoid requiring TyP by using the internally definable universe of homotopy propositions `hProp` [23]. If the relation is `hProp`-valued, terms respect reflexivity proofs up to the internal identity type. However to show that the relation for Π types is in `hProp`, we need that `hProp` is closed under Π . To prove this, we need function extensionality, which defeats the purpose of the model bootstrapping function extensionality.

In this paper we show that in intensional type theory there is an alternative notion of proposition that is closed under Π . A type A is an `hProp` if any two elements of A are equal. We can also express this equation in a point-free way: the two functions “first” and “second” both having type $A \rightarrow A \rightarrow A$ are equal. We call this property `isPfProp` for point-free propositions.

$$\text{isHProp } A \equiv (a a' : A) \rightarrow \text{Id}_A a a' \qquad \text{isPfProp } A \equiv \text{Id}_{(A \rightarrow A \rightarrow A)} (\lambda a a'. a) (\lambda a a'. a')$$

In the presence of function extensionality, `isHProp` A and `isPfProp` A are equivalent. However, in intensional type theory without function extensionality, the latter is stronger. `isPfProp` classifies definitionally proof irrelevant types in the empty context: from canonicity it follows that if `isPfProp` A for a closed type A , then all elements of A are definitionally equal. For a type family $A : D \rightarrow \text{Type}$ over a closed type we use a dependent variant of `isPfProp`:

$$\text{isPfPropd } A \equiv \text{Id}_{((d:D) \rightarrow A \rightarrow A \rightarrow A)} (\lambda d a a'. a) (\lambda d a a'. a')$$

In intensional type theory, unlike `hProp`, `isPfPropd` is closed under Π types. This essentially relies on the η rule for Π types. Using η for Σ and \top , we can prove that `isPfPropd` is closed under these type formers too. `isPfProp` only includes \perp if it has a weak η rule saying that any two elements of \perp are definitionally equal. This is usually not the case in intensional type theory where \perp is defined as an inductive type.

With the help of point-free propositions, we give a partial positive answer to Martin-Löf’s question: in intensional type theory without a sort of propositions, we define the setoid model with \perp , \top , Π , Σ , types, a sort TyP closed under \top , Π , Σ and a TyP-valued identity type with function extensionality. Our answer is partial because \perp is not in TyP, and the model does not support inductive types, or a universe of propositions. We also define an external version of this model as a model construction taking as input a model of intensional type theory, and outputting a model with extensionality principles. This latter construction only uses external point-free propositions which are the same as subobjects in category theory, but we still haven’t encountered it in the literature.

Recently, there is a renewed interest in models of type theory with a sort TyP. Agda was extended with a universe of strict propositions [13], this was used to formalise fully featured variants of the setoid model [4, 3, 18], strict presheaf models were built using TyP [22], and the metatheory of type theories with TyP was studied [1, 11]. One difference between TyP and `pfProp` is that (as every sort) the former is static: it only includes types which are built

into it. The latter is dynamic: any type is included for which all elements are definitionally equal. Another difference is that proof irrelevance holds definitionally for assumed elements of TyP , while we only know proof irrelevance up to propositional equality for members of pfProp .

More generally, in intensional type theory, point-free equations can be used to describe strict algebraic structures. One has to express the algebraic equations in a point-free way. For example, in a strict monoid with carrier M and binary operation $- \otimes -$, associativity is expressed as $\text{Id}_{M \rightarrow M \rightarrow M \rightarrow M} (\lambda x y z. (x \otimes y) \otimes z) (\lambda x y z. x \otimes (y \otimes z))$. Natural numbers with addition are not a strict monoid because addition is only weakly associative. An example of a strict monoid is the function space $A \rightarrow A$ for any type A with composition as the binary operation.

1.1 Structure of the paper

After describing related work, in Section 2 we explain our notation and the notion of model of type theory we use (category with families). In Section 3 we define point-free propositions and show that they are closed under \top , Σ and Π . In Section 4, we show that any model of type theory can be equipped with a sort of strict propositions. This can be seen as the external version of Section 3. We compare the internal and external notions of propositions in Section 5. Then we describe how point-free propositions can be applied to construct the setoid model. As a warmup, we define the model construction externally (Section 6). Then we turn to our main application of internal point-free propositions and define the setoid model internally to a model of intensional type theory (Section 7). In Section 8 we give more examples of strict algebraic structures. We conclude in Section 9.

Sections 3 and 7 were formalised in Agda [12], and can be understood without much intuition about categories with families.

1.2 Related work

Hofmann defined two versions of the setoid model in an intensional metatheory [15], one of them did not have dependent types, the other justified some computation rules (e.g. β rules for Σ types) only up to propositional equality, and not definitionally. Altenkirch [2] justified all the rules of type theory but relied on a definitionally proof-irrelevant universe of propositions. He sketched a normalisation proof for a type theory with such a universe. Coquand [9] defined a setoid model in intensional type theory which justifies a weak function space: there is no substitution rule for λ and no η rule. Palmgren [21] formalised a set-theoretic interpretation of extensional type theory in an intensional metatheory. He used setoids for encoding sets as well-founded trees quotiented by bisimulation, hence it can also be seen as a setoid model. Thus it is similar to our Construction 17 and it justifies more types including inductive types and a universe. It is not clear whether one can obtain a model construction analogous to Construction 15 from his interpretation.

Strict propositions were introduced in Agda and Coq in a way that is compatible with univalence [13]. Issues with rewriting-style normalisation for a type theory with strict propositions, a strict identity type and a strong transport were found by Abel and Coquand [1]. Normalisation for type theory with strict propositions but without such an identity type was proved by Coquand [11].

The setoid model as a model construction was described in [4] together with an Agda formalisation using strict propositions in Agda. This was extended with an inductive-recursive universe of setoids in [3].

In [4], a variant of the setoid model was described in which transport has a definitional computation rule. In the accompanying formalisation, a point-free equation was used to ensure this property: instead of $(a : A) \rightarrow \text{coe}_A \text{refl } a = a$, the equation $(\lambda a. \text{coe}_A \text{refl } a) = (\lambda a. a)$ was used. In his brilliant paper [16], Hugunin shows that function extensionality is not needed to define natural numbers (and inductive types) from W-types in intensional type theory. He constructs a predicate which selects the “canonical” elements in natural numbers defined by W-types. His construction has a similar “point-free” flavour and also essentially relies on η for function space.

2 Type theory

Our metatheory is extensional type theory and we use notations similar to Agda’s. We write Type for the Russell universe (we don’t write levels explicitly, but we work in a predicative setting), we write \equiv for definitional equality, we write $(x : A) \rightarrow B$ for function space with $\lambda(x : A).t$ or $\lambda x.t$ for abstraction, juxtaposition for application, $(x : A) \times B$ for Σ types with a, b for pairing and $\pi_1 ab, \pi_2 ab$ for projections. We use the lower case Simonyi naming convention, e.g. ab is a name for a variable in $(x : A) \times B$. We use implicit arguments and implicit quantifications which we sometimes specify explicitly in subscripts. We write \top , tt for the unit type and its constructor. Function space, dependent products and unit have η laws. We write $\text{Id}_A a a'$ or $a =_A a'$ or simply $a = a'$ for the identity type, it has constructor $\text{refl} : \text{Id}_A a a$ and eliminator $\text{J} : (P : (a' : A) \rightarrow a = a' \rightarrow \text{Type}) \rightarrow P a \text{refl} \rightarrow (e : a = a') \rightarrow P a' e$ with definitional computation rules. We write $\text{transp} : (P : A \rightarrow \text{Type}) \rightarrow a = a' \rightarrow P a \rightarrow P a'$, $e \blacksquare e' : a = a''$ for $e : a = a'$ and $e' : a' = a''$, $\text{ap } f e : f a = f a'$ for $e : a = a'$, all defined via J . The empty type is denoted \perp with eliminator elim_\perp . We assume quotient inductive-inductive types (QIITs), that is, we have syntaxes for type theories (see paragraph after the next one).

In some places (e.g. in sections 3 and 7), we work internally to a model of intensional type theory, and use the same notations as for our metatheory. In these cases we specify precisely what features our model has and we only use those features, for example we don’t use equality reflection. In such cases we use the phrase “external” to refer to the metatheory.

The notion of model of type theory we use is category with families (CwF, [8]). Using this presentation, type theory is a generalised algebraic theory and the syntax of a type theory is the initial algebra which is a QIIT. In extensional type theory, it is enough to assume the existence of a single QIIT to obtain syntaxes for all generalised algebraic theories [17]. We assume the existence of this QIIT (called the theory of QIIT signatures in [17]).

We give some intuition for the description of type theory as a CwF here. A gentler introduction is e.g. [5]. Figure 1 lists the components of a model of type theory with \top , Σ , Π , \perp and Id types. A model of type theory consists of a category with families (CwF, left hand side of the figure), that is, a category of contexts and substitutions $(\text{Con}, \dots, \text{idr})$ with a terminal object (the empty context \blacklozenge , the empty substitution $\epsilon, \blacklozenge\eta$), a presheaf of types $(\text{Ty}, \dots, [\text{id}])$ and a locally representable dependent presheaf of terms over types $(\text{Tm}, \dots, \triangleright\eta)$. Local representability is also called comprehension, and consists of the context extension operation $-\triangleright-$ together with the natural isomorphism $\text{Sub } \Delta (\Gamma \triangleright A) \cong (\gamma : \text{Sub } \Delta \Gamma) \times \text{Tm } \Delta (A[\gamma])$ witnessed by $-, -, \dots, \triangleright\eta$. Note that many operations have implicit arguments, for example $-\circ-$ takes Γ, Δ, Θ implicitly. Also, some equations only typecheck because of previous equations, for example, $[\text{id}]$ for terms depends on $[\text{id}]$ for types: the left hand side is in $\text{Tm } \Gamma (A[\text{id}])$, the right hand side is in $\text{Tm } \Gamma A$. We don’t write the transports because we work in extensional type theory.

Con	: Set	\top	: $\text{Ty } \Gamma$
Sub	: $\text{Con} \rightarrow \text{Con} \rightarrow \text{Set}$	$\top[]$: $\top[\gamma] \equiv \top$
$-\circ-$: $\text{Sub } \Delta \Gamma \rightarrow \text{Sub } \Theta \Delta \rightarrow \text{Sub } \Theta \Gamma$	tt	: $\text{Tm } \Gamma \top$
ass	: $(\gamma \circ \delta) \circ \theta \equiv \gamma \circ (\delta \circ \theta)$	$\top\eta$: $(t : \text{Tm } \Gamma \top) \rightarrow t \equiv \text{tt}$
id	: $\text{Sub } \Gamma \Gamma$	Σ	: $(A : \text{Ty } \Gamma) \rightarrow \text{Ty } (\Gamma \triangleright A) \rightarrow \text{Ty } \Gamma$
idl	: $\text{id} \circ \gamma \equiv \gamma$	$\Sigma[]$: $(\Sigma A B)[\gamma] \equiv \Sigma (A[\gamma]) (B[\gamma \circ \mathfrak{p}, \mathfrak{q}])$
idr	: $\gamma \circ \text{id} \equiv \gamma$	$-, -$: $(a : \text{Tm } \Gamma A) \rightarrow \text{Tm } \Gamma (B[\text{id}, a]) \rightarrow \text{Tm } \Gamma (\Sigma A B)$
\blacklozenge	: Con	π_1	: $\text{Tm } \Gamma (\Sigma A B) \rightarrow \text{Tm } \Gamma A$
ϵ	: $\text{Sub } \Gamma \blacklozenge$	π_2	: $(ab : \text{Tm } \Gamma (\Sigma A B)) \rightarrow \text{Tm } \Gamma (B[\text{id}, \pi_1 ab])$
$\blacklozenge\eta$: $(\sigma : \text{Sub } \Gamma \blacklozenge) \rightarrow \sigma \equiv \epsilon$	$\Sigma\beta_1$: $\pi_1 (a, b) \equiv a$
Ty	: $\text{Con} \rightarrow \text{Set}$	$\Sigma\beta_2$: $\pi_2 (a, b) \equiv b$
$-[-]$: $\text{Ty } \Gamma \rightarrow \text{Sub } \Delta \Gamma \rightarrow \text{Ty } \Delta$	$\Sigma\eta$: $(\pi_1 ab, \pi_2 ab) \equiv ab$
$[\circ]$: $A[\gamma \circ \delta] \equiv A[\gamma][\delta]$	$, []$: $(a, b)[\gamma] \equiv (a[\gamma], b[\gamma])$
$[\text{id}]$: $A[\text{id}] \equiv A$	Π	: $(A : \text{Ty } \Gamma) \rightarrow \text{Ty } (\Gamma \triangleright A) \rightarrow \text{Ty } \Gamma$
Tm	: $(\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Set}$	$\Pi[]$: $(\Pi A B)[\gamma] \equiv \Pi (A[\gamma]) (B[\gamma \circ \mathfrak{p}, \mathfrak{q}])$
$-[-]$: $\text{Tm } \Gamma A \rightarrow (\gamma : \text{Sub } \Delta \Gamma) \rightarrow \text{Tm } \Delta (A[\gamma])$	lam	: $\text{Tm } (\Gamma \triangleright A) B \rightarrow \text{Tm } \Gamma (\Pi A B)$
$[\circ]$: $a[\gamma \circ \delta] \equiv a[\gamma][\delta]$	app	: $\text{Tm } \Gamma (\Pi A B) \rightarrow \text{Tm } (\Gamma \triangleright A) B$
$[\text{id}]$: $a[\text{id}] \equiv a$	$\Pi\beta$: $\text{app} (\text{lam } t) \equiv t$
$-\triangleright-$: $(\Gamma : \text{Con}) \rightarrow \text{Ty } \Gamma \rightarrow \text{Con}$	$\Pi\eta$: $\text{lam} (\text{app } t) \equiv t$
$-, -$: $(\gamma : \text{Sub } \Delta \Gamma) \rightarrow \text{Tm } \Delta (A[\gamma]) \rightarrow \text{Sub } \Delta (\Gamma \triangleright A)$	$\text{lam}[]$: $(\text{lam } t)[\gamma] \equiv \text{lam} (t[\gamma \circ \mathfrak{p}, \mathfrak{q}])$
\mathfrak{p}_A	: $\text{Sub } (\Gamma \triangleright A) \Gamma$	\perp	: $\text{Ty } \Gamma$
\mathfrak{q}_A	: $\text{Tm } (\Gamma \triangleright A) (A[\mathfrak{p}])$	$\perp[]$: $\perp[\gamma] \equiv \perp$
$\triangleright\beta_1$: $\mathfrak{p} \circ (\gamma, a) \equiv \gamma$	elim_\perp	: $\text{Tm } \Gamma \perp \rightarrow \text{Tm } \Gamma A$
$\triangleright\beta_2$: $\mathfrak{q}[\gamma, a] \equiv a$	$\text{elim}_\perp[]$: $(\text{elim}_\perp t)[\gamma] \equiv \text{elim}_\perp (t[\gamma])$
$\triangleright\eta$: $(\mathfrak{p} \circ \gamma a, \mathfrak{q}[\gamma a]) \equiv \gamma a$	Id_-	: $(A : \text{Ty } \Gamma) \rightarrow \text{Tm } \Gamma A \rightarrow \text{Tm } \Gamma A \rightarrow \text{Ty } \Gamma$
		$\text{Id}[]$: $(\text{Id}_A a a')[\gamma] \equiv \text{Id}_{A[\gamma]} (a[\gamma]) (a'[\gamma])$
		refl	: $\text{Tm } \Gamma (\text{Id}_A a a)$
		$\text{refl}[]$: $\text{refl}[\gamma] \equiv \text{refl}$
		J	: $(P : \text{Ty } (\Gamma \triangleright A \triangleright \text{Id}_{A[\mathfrak{p}]} (a[\mathfrak{p}]) \mathfrak{q})) \rightarrow \text{Tm } \Gamma (P[\text{id}, a, \text{refl}]) \rightarrow (e : \text{Tm } \Gamma (\text{Id}_A a a')) \rightarrow \text{Tm } \Gamma (P[\text{id}, a', e])$
		$\text{J}\beta$: $\text{J } P w \text{ refl} \equiv w$
		$\text{J}[]$: $(\text{J } P w e)[\gamma] \equiv \text{J} (P[\gamma \circ \mathfrak{p} \circ \mathfrak{p}, \mathfrak{q}[\mathfrak{p}], \mathfrak{q}]) (w[\gamma]) (e[\gamma])$

■ **Figure 1** A model of type theory with \top , Σ , Π , \perp , Id . The left column is the definition of CwF , the right column contains the rules for the type formers, one after the other, in the same order.

6:6 Internal Strict Propositions Using Point-Free Equations

$$\begin{aligned}
, \circ & : (\gamma, a) \circ \delta \equiv (\gamma \circ \delta, a[\delta]) \\
\pi_1[] & : (\pi_1 ab)[\gamma] \equiv \pi_1(ab[\gamma]) \\
\pi_2[] & : (\pi_2 ab)[\gamma] \equiv \pi_2(ab[\gamma]) \\
- \times - & : \mathbf{Ty} \Gamma \rightarrow \mathbf{Ty} \Gamma \rightarrow \mathbf{Ty} \Gamma \\
A \times B & := \Sigma A (B[p]) \\
\mathbf{app}[] & : (\mathbf{app} t)[\gamma \circ p, q] \equiv \mathbf{app} (t[\gamma]) \\
- \$ - & : \mathbf{Tm} \Gamma (\Pi A B) \rightarrow (a : \mathbf{Tm} \Gamma A) \rightarrow \mathbf{Tm} \Gamma (B[\mathbf{id}, a]) \\
t \$ a & := (\mathbf{app} t)[\mathbf{id}, a] \\
\$ \beta & : \mathbf{lam} t \$ a \equiv t[\mathbf{id}, a] \\
\$ [] & : (t \$ a)[\gamma] \equiv t[\gamma] \$ a[\gamma] \\
- \Rightarrow - & : \mathbf{Ty} \Gamma \rightarrow \mathbf{Ty} \Gamma \rightarrow \mathbf{Ty} \Gamma \\
A \Rightarrow B & := \Pi A (B[p])
\end{aligned}$$

■ **Figure 2** Provable equations and definable operations in a model of type theory with Σ , Π .

$$\begin{aligned}
\mathbf{TyP} & : \mathbf{Con} \rightarrow \mathbf{Set} \\
-[-] & : \mathbf{TyP} \Gamma \rightarrow \mathbf{Sub} \Delta \Gamma \rightarrow \mathbf{TyP} \Delta \\
[\circ] & : A[\gamma \circ \delta] \equiv A[\gamma][\delta] \\
[\mathbf{id}] & : A[\mathbf{id}] \equiv A \\
\uparrow & : \mathbf{TyP} \Gamma \rightarrow \mathbf{Ty} \Gamma \\
\uparrow [] & : (\uparrow A)[\gamma] \equiv \uparrow(A[\gamma]) \\
\mathbf{irr} & : (u v : \mathbf{Tm} \Gamma (\uparrow A)) \rightarrow u \equiv v \\
\mathbf{TP} & : \mathbf{TyP} \Gamma \\
\mathbf{TP}[] & : \mathbf{TP}[\gamma] \equiv \mathbf{TP} \\
\mathbf{ttP} & : \mathbf{Tm} \Gamma \mathbf{TP} \\
\Sigma \mathbf{P} & : (A : \mathbf{TyP} \Gamma) \rightarrow \mathbf{TyP} (\Gamma \triangleright \uparrow A) \rightarrow \mathbf{TyP} \Gamma \\
\Sigma \mathbf{P}[] & : (\Sigma \mathbf{P} A B)[\gamma] \equiv \Sigma \mathbf{P} (A[\gamma]) (B[\gamma \circ p, q]) \\
-, \mathbf{P} - & : (a : \mathbf{Tm} \Gamma (\uparrow A)) \rightarrow \mathbf{Tm} \Gamma (\uparrow B[\mathbf{id}, a]) \rightarrow \mathbf{Tm} \Gamma (\uparrow \Sigma \mathbf{P} A B) \\
\pi_1 \mathbf{P} & : \mathbf{Tm} \Gamma (\uparrow \Sigma \mathbf{P} A B) \rightarrow \mathbf{Tm} \Gamma A \\
\pi_2 \mathbf{P} & : (ab : \mathbf{Tm} \Gamma (\uparrow \Sigma \mathbf{P} A B)) \rightarrow \mathbf{Tm} \Gamma (\uparrow B[\mathbf{id}, \pi_1 ab]) \\
\Pi \mathbf{P} & : (A : \mathbf{Ty} \Gamma) \rightarrow \mathbf{TyP} (\Gamma \triangleright A) \rightarrow \mathbf{TyP} \Gamma \\
\Pi \mathbf{P}[] & : (\Pi \mathbf{P} A B)[\gamma] \equiv \Pi \mathbf{P} (A[\gamma]) (B[\gamma \circ p, q]) \\
\mathbf{lamP} & : \mathbf{Tm} (\Gamma \triangleright A) (\uparrow B) \rightarrow \mathbf{Tm} \Gamma (\uparrow \Pi \mathbf{P} A B) \\
\mathbf{appP} & : \mathbf{Tm} \Gamma (\uparrow \Pi \mathbf{P} A B) \rightarrow \mathbf{Tm} (\Gamma \triangleright A) (\uparrow B)
\end{aligned}$$

■ **Figure 3** A model of type theory has a sort of proof-irrelevant propositions closed under \top , Σ , Π .

Variables are represented using typed De Bruijn indices. The zero De Bruijn index is $q : \mathsf{Tm}(\Gamma \triangleright A)(A[p])$, the one index is given by $q[p] : \mathsf{Tm}(\Gamma \triangleright A \triangleright B)(A[p][p])$, two is $q[p][p] : \mathsf{Tm}(\Gamma \triangleright A \triangleright B \triangleright C)(A[p][p][p])$, and so on. Some provable equations and definable operations are listed in Figure 2.

The right hand side of Figure 1 lists rules for \top , Σ , Π , \perp and Id types, in this order. The first three type formers have η laws, the latter two don't (they are instances of inductive types). Every operation comes with substitution laws (e.g. $\mathsf{lam}[]$), some of them are not listed as they are provable (see Figure 2). Non-dependent special cases of Π and Σ are also listed there.

Figure 3 lists the operations and equations for a model having a sort of definitionally proof-irrelevant propositions TyP which is closed under \top , Σ and Π . Terms of propositional types are expressed with the help of lifting \uparrow which converts a TyP into a Ty . Because of irr , there is no need to state equations for terms of lifted types, all equations hold.

Two important properties of models that we sometimes assume are canonicity [10] and normalisation [6, 10]. Canonicity for \perp says that there is no $\mathsf{Tm} \blacklozenge \perp$. Canonicity for Id says that for any $t : \mathsf{Tm} \blacklozenge (\mathsf{Id}_A a a')$, we have $a \equiv a'$ and $t \equiv \mathsf{refl}$. Normalisation says that there is a function from terms to normal forms $\mathsf{norm} : \mathsf{Tm} \Gamma A \rightarrow \mathsf{Nf} \Gamma A$ such that all terms are equal to their normalised versions (\uparrow^{-1} is the inclusion from Nf to Tm): for all $a : \mathsf{Tm} \Gamma A$, $\uparrow \mathsf{norm} a = a$. Normal forms for the theory of Figure 1 are defined mutually with variables and neutral terms by the following three inductive types.

$x ::= q \mid x[p]$	variables
$n ::= x \mid \pi_1 n \mid \pi_2 n \mid n \$ v \mid \mathsf{elim}_{\perp} n \mid \mathsf{J} A v n$	neutral terms
$v ::= n^* \mid \mathsf{tt} \mid (v, v) \mid \mathsf{lam} v \mid \mathsf{refl}$	normal forms

These should be understood as typed rules and there is a restriction (n^*) that only neutral terms at base types are included in normal forms. Base types are \perp and Id in our case.

Sometimes we just talk about intensional type theory when we don't want to specify precisely what type formers we have in a model.

3 Point-free propositions internally

In this section we show that (the dependent variant of) point-free propositions is closed under \top , Σ and Π . We work internally to a model of type theory with a universe Type closed under type formers Id , \top , Σ , Π . This section was formalised in Agda [12].

The η rule for \top says that for any two $t, t' : \top$ we have $t \equiv t'$, so we also have that $(\lambda(t t' : \top).t) \equiv (\lambda t t'.t')$, hence $\mathsf{refl} : ((\lambda t t'.t) = (\lambda t t'.t')) \equiv \mathsf{isPfProp} \top$.

As a warmup for Σ , we prove closure under non-dependent products.

► **Proposition 1.** *If $\mathsf{isPfProp} A$ and $\mathsf{isPfProp} B$, then $\mathsf{isPfProp} (A \times B)$.*

Proof. We assumed $p_A : \mathsf{isPfProp} A \equiv ((\lambda(a a' : A).a) = (\lambda a a'.a'))$ and $p_B : \mathsf{isPfProp} B \equiv ((\lambda(b b' : B).b) = (\lambda b b'.b'))$ and we want to obtain that $A \times B$ is a point-free proposition.

$$\begin{aligned} p_{A \times B} : \mathsf{isPfProp} (A \times B) &\equiv ((\lambda ab ab'.ab) = (\lambda ab ab'.ab')) \equiv \\ &((\lambda ab ab'.(\pi_1 ab, \pi_2 ab)) = (\lambda ab ab'.(\pi_1 ab', \pi_2 ab'))) \end{aligned}$$

6:8 Internal Strict Propositions Using Point-Free Equations

When rewriting the type of $p_{A \times B}$, we applied the η rule for products which says that $ab \equiv (\pi_1 ab, \pi_2 ab)$ for any $ab : A \times B$. Then we prove the equality in two steps: first we use p_A to show that $\pi_1 ab = \pi_1 ab'$ while we keep the $\pi_2 ab$ component constant

$$\begin{aligned} p_{A \times B}^1 &: (\lambda ab ab'. (\pi_1 ab, \pi_2 ab)) = (\lambda ab ab'. (\pi_1 ab', \pi_2 ab)) \\ p_{A \times B}^1 &:\equiv \mathbf{ap} (\lambda z. \lambda ab ab'. (z (\pi_1 ab) (\pi_1 ab'), \pi_2 ab)) p_A, \end{aligned}$$

then we use p_B to show that $\pi_2 ab = \pi_2 ab'$ while we keep the $\pi_1 ab'$ components constant. In the middle we have the function returning the mixed pair $(\pi_1 ab', \pi_2 ab)$.

$$\begin{aligned} p_{A \times B}^2 &: (\lambda ab ab'. (\pi_1 ab', \pi_2 ab)) = (\lambda ab ab'. (\pi_1 ab', \pi_2 ab')) \\ p_{A \times B}^2 &:\equiv \mathbf{ap} (\lambda z. \lambda ab ab'. (\pi_1 ab', z (\pi_2 ab) (\pi_2 ab'))) p_B \end{aligned}$$

We obtain the desired equality via transitivity:

$$p_{A \times B} :\equiv p_{A \times B}^1 \blacksquare p_{A \times B}^2 \quad \blacktriangleleft$$

To show closure of point-free propositions under Σ types, we have $A : \mathbf{Type}$, $B : A \rightarrow \mathbf{Type}$, $\mathbf{isPfProp} A$, but assuming $(a : A) \rightarrow \mathbf{isPfProp} (B a)$ is not enough. We express that B is a family of propositions using a dependent version of $\mathbf{isPfProp}$:

$$\begin{aligned} \mathbf{isPfPropd} &: (A \rightarrow \mathbf{Type}) \rightarrow \mathbf{Type} \\ \mathbf{isPfPropd} B &:\equiv (\lambda (a : A) (b b' : B a). b) = (\lambda a b b'. b') \end{aligned}$$

The non-dependent version is a special case when there is an element of the indexing type $a_0 : A$, because given $B : \mathbf{Type}$ and $p_B : \mathbf{isPfPropd} (\lambda (a : A). B)$, we have $\mathbf{ap} (\lambda z. z a_0) p_B : \mathbf{isPfProp} B$.

We show the dependent version of closure under Σ types.

► **Proposition 2.** *Given $A : D \rightarrow \mathbf{Type}$ and $B : \Sigma D A \rightarrow \mathbf{Type}$, if $\mathbf{isPfPropd} A$ and $\mathbf{isPfPropd} B$, then $\mathbf{isPfPropd} (\lambda d. \Sigma (A d) (\lambda a. B (d, a)))$.*

Proof. We have $p_A : \mathbf{isPfPropd} A \equiv (\lambda d a a'. a) = (\lambda d a a'. a')$ and $p_B : \mathbf{isPfPropd} B \equiv (\lambda d a b b'. b) = (\lambda d a b b'. b')$, our goal is to obtain

$$p_{\Sigma AB} : (\lambda d ab ab'. ab) = (\lambda d ab ab'. ab') \equiv (\lambda d ab ab'. (\pi_1 ab, \pi_2 ab)) = (\lambda d ab ab'. (\pi_1 ab', \pi_2 ab')).$$

We want to prove this in two steps as for non-dependent products, but because B depends on A , the middle pair $(\pi_1 ab', \pi_2 ab)$ is not well-typed. We replace the second component $\pi_2 ab : B (d, \pi_1 ab)$ with

$$\mathbf{transp}_{\lambda a. B (d, a)} \left(\mathbf{ap} (\lambda z. z d (\pi_1 ab) (\pi_1 ab')) p_A \right) (\pi_2 ab) : B (d, \pi_1 ab'),$$

and we will use a more general version of this second component defined as

$$f ab ab' e := \mathbf{transp}_{\lambda a. B (d, a)} \left(\mathbf{ap} (\lambda z. z d (\pi_1 ab) (\pi_1 ab')) e \right) (\pi_2 ab) : B (d, h d (\pi_1 ab) (\pi_1 ab'))$$

for any d, ab, ab', h and $e : (\lambda d a a'. a) = h$. Now the first step has type

$$p_{\Sigma AB}^1 : (\lambda d ab ab'. ab) = (\lambda d ab ab'. (\pi_1 ab', f ab ab' p_A))$$

and we prove it by induction on p_A using \mathbf{J} :

$$p_{\Sigma AB}^1 :\equiv \mathbf{J} \left(\lambda h e. (\lambda d ab ab'. ab) = (\lambda d ab ab'. (h d (\pi_1 ab) (\pi_1 ab'), f ab ab' e)) \right) \mathbf{refl} p_A$$

In the next step we simply use ap on p_B and we conclude by transitivity:

$$\begin{aligned} p_{\Sigma AB}^2 &: (\lambda d \text{ ab } ab'. (\pi_1 \text{ ab}', f \text{ ab } ab' p_A)) = (\lambda d \text{ ab } ab'. ab') \\ p_{\Sigma AB}^2 &\equiv \text{ap} \left(\lambda z. \lambda d \text{ ab } ab'. (\pi_1 \text{ ab}', z (d, \pi_1 \text{ ab}') (f \text{ ab } ab' p_A) (\pi_2 \text{ ab}')) \right) p_B \\ p_{\Sigma AB} &\equiv p_{\Sigma AB}^1 \blacksquare p_{\Sigma AB}^2 \quad \blacktriangleleft \end{aligned}$$

► **Corollary 3.** For $A : \text{Type}$ and $B : A \rightarrow \text{Type}$, if $\text{isPfProp } A$ and $\text{isPfPropd } B$, then $\text{isPfProp } (\Sigma A B)$.

Finally, we show closure of isPfPropd under dependent function space.

► **Proposition 4.** Given $A : D \rightarrow \text{Type}$, $B : \Sigma D A \rightarrow \text{Type}$, if $\text{isPfPropd } B$, then $\text{isPfPropd } (\lambda d. (a : A d) \rightarrow B (d, a))$.

Proof. Using $p_B : (\lambda da \text{ bb}'. b) = (\lambda da \text{ bb}'. b')$, we define

$$\begin{aligned} p_{\Pi AB} &: (\lambda d f f'. f) = (\lambda d f f'. f') \equiv (\lambda d f f' a. f a) = (\lambda d f f' a. f' a) \\ p_{\Pi AB} &\equiv \text{ap} (\lambda z d f f' a. z (d, a) (f a) (f' a)) p_B. \quad \blacktriangleleft \end{aligned}$$

► **Corollary 5.** For $A : \text{Type}$ and $B : A \rightarrow \text{Type}$, if $\text{isPfPropd } B$, then $\text{isPfProp } ((a : A) \rightarrow B a)$.

4 Point-free propositions externally

In this section we show that any model of type theory with \top , Σ , Π types has a sort TyP closed under the same type formers. This can be seen as an externalisation of the previous section.

Recall that a model of type theory (a CwF , see Section 2) has a sort of strict propositions if there is a presheaf TyP together with a “lifting” natural transformation \uparrow into Ty , and terms of a lifted type are equal.

First we define a predicate on types expressing externally that the type is a point-free proposition.

► **Definition 6.** For a type $A : \text{Ty } \Gamma$ in any CwF , let $\text{isExtPfProp } A := (\mathfrak{q}_A[\mathfrak{p}_{A[\mathfrak{p}]}] \equiv \mathfrak{q}_{A[\mathfrak{p}]})$.

That is, in the context $\Gamma \triangleright A \triangleright A[\mathfrak{p}]$, the terms $\mathfrak{q}[\mathfrak{p}]$ and \mathfrak{q} (1 and 0 De Bruijn indices, both having type $A[\mathfrak{p}][\mathfrak{p}]$) are definitionally equal. We call this the external variant of pfProp because it is clear that it is equivalent to saying $\text{lam} (\text{lam } (\mathfrak{q}[\mathfrak{p}])) \equiv \text{lam} (\text{lam } \mathfrak{q})$ which is the external statement of $\lambda x y. x = \lambda x y. y$. In the next section, we will relate the external and internal variants formally.

Elements of a type which isExtPfProp are equal in any context.

► **Proposition 7.** For a type A , $\text{isExtPfProp } A$ is equivalent to

$$u \equiv v \text{ for all } \gamma : \text{Sub } \Delta \Gamma \text{ and } u, v : \text{Tm } \Delta (A[\gamma]).$$

Proof. Left to right: we have $\mathfrak{q}[\mathfrak{p}][\gamma, u, v] \equiv \mathfrak{q}[\gamma, u, v]$, hence $u \equiv v$. Right to left: we choose $u := \mathfrak{q}[\mathfrak{p}]$, $v := \mathfrak{q}$. ◀

In category theory, external point-free propositions over Γ are called subobjects of Γ .

► **Proposition 8.** For an $A : \text{Ty } \Gamma$, $\text{isExtPfProp } A$ is equivalent to the morphism $\mathfrak{p}_A : \text{Sub } (\Gamma \triangleright A) \Gamma$ being a monomorphism.

6:10 Internal Strict Propositions Using Point-Free Equations

Proof. Left to right: given $p_A \circ (\gamma, a) \equiv p_A \circ (\gamma', a')$, we need to show $(\gamma, a) \equiv (\gamma', a')$. Using the assumption we have $\gamma \equiv p_A \circ (\gamma, a) \equiv p_A \circ (\gamma', a') \equiv \gamma'$, hence a and a' are both in $\text{Tm } \Delta (A[\gamma])$. We get $a \equiv a'$ from Proposition 7.

Right to left: given two terms $a, a' : \text{Tm } \Gamma (A[\gamma])$, we have $p_A \circ (\gamma, a) \equiv \gamma \equiv p_A \circ (\gamma, a')$, hence by assumption $(\gamma, a) \equiv (\gamma, a')$ and applying $q[-]$ to both sides we obtain $a \equiv a'$. \blacktriangleleft

► **Construction 9.** Every CwF with \top, Σ, Π can be equipped with a sort of strict propositions closed under the same type formers.

Construction. We have to define all components in Figure 3. We define

$$\text{TyP } \Gamma := (A : \text{Ty } \Gamma) \times \text{isExtPfProp } A.$$

Substitution is defined by ordinary type substitution of the first component and the equation for substituted types holds by the following argument.

$$\begin{aligned} q_{A[\gamma]}[p_{A[\gamma][p]}] & \equiv ([\circ], \triangleright\beta_1, \triangleright\beta_2) \\ q_A[p_{A[p]}][\gamma \circ p \circ p, q_{A[\gamma]}[p_{A[\gamma][p]}], q_{A[\gamma]}[p]] & \equiv (\text{assumption}) \\ q_{A[p]}[\gamma \circ p \circ p, q_{A[\gamma]}[p_{A[\gamma][p]}], q_{A[\gamma]}[p]] & \equiv (\triangleright\beta_2) \\ q_{A[\gamma][p]} & \end{aligned}$$

The \uparrow operation is defined by $\uparrow(A, p_A) := A$. Irrelevance holds by Proposition 7. $\top P$ is defined as \top and $\text{isExtPfProp } \top$ holds by $\top\eta$. We define $\Sigma P (A, p_A) (B, p_B)$ by $(\Sigma A B, p_{\Sigma A B})$ where $p_{\Sigma A B}$ is proven using Proposition 7 for $u, v : \text{Tm } \Delta (\Sigma A B[\gamma])$ by

$$u \stackrel{\Sigma\eta}{\equiv} (\pi_1 u, \pi_2 u) \stackrel{p_A, p_B}{\equiv} (\pi_1 v, \pi_2 v) \stackrel{\Sigma\eta}{\equiv} v.$$

We define $\Pi P A (B, p_B)$ by $(\Pi A B, p_{\Pi A B})$ where $p_{\Pi A B}$ is proven using Proposition 7 for $u, v : \text{Tm } \Delta (\Pi A B[\gamma])$ by

$$u \stackrel{\Pi\eta}{\equiv} \text{lam}(\text{app } u) \stackrel{p_B}{\equiv} \text{lam}(\text{app } v) \stackrel{\Pi\eta}{\equiv} v. \quad \blacktriangleleft$$

5 Relationship of different notions of being a proposition

For a type family $A : D \rightarrow \text{Type}$, being a family of homotopy propositions and a family of point-free propositions were defined internally as follows.

$$\begin{aligned} \text{isHPropd } A & \equiv (d : D)(a a' : A d) \rightarrow \text{Id}_{(A d)} a a' \\ \text{isPfPropd } A & \equiv \text{Id}_{(d:D) \rightarrow A d \rightarrow A d \rightarrow A d} (\lambda d a a'. a) (\lambda d a a'. a') \end{aligned}$$

Externally, these can be seen as the following two elements of $\text{Ty } \blacklozenge$ for $A : \text{Ty } (\blacklozenge \triangleright D)$. We also repeat the definition of isExtPfProp for comparison which is a metatheoretic equality.

$$\begin{aligned} \text{isHPropd } A & \equiv \Pi D \left(\Pi A \left(\Pi (A[p]) \left(\text{Id}_{A[p][p]} (q[p]) q \right) \right) \right) \\ \text{isPfPropd } A & \equiv \text{Id}_{\Pi D (A \Rightarrow A \Rightarrow A)} \left(\text{lam} \left(\text{lam} \left(\text{lam} (q[p]) \right) \right) \right) \left(\text{lam} \left(\text{lam} \left(\text{lam } q \right) \right) \right) \\ \text{isExtPfProp } A & \equiv (q[p] \equiv q) \quad \left(\text{both sides in } \text{Tm } (\blacklozenge \triangleright D \triangleright A \triangleright A[p]) (A[p][p]) \right) \end{aligned}$$

We first compare internal point-free propositions and external ones. They coincide for a type where we collect all dependencies into a single closed type D .

► **Proposition 10.** *In a model of type theory with Π , Id and canonicity, given an $A : \text{Ty}(\diamond \triangleright D)$, there is a $\text{Tm} \diamond (\text{isPfProp} A)$ if and only if $\text{isExtPfProp} A$.*

Proof. Right to left: if $q[p] \equiv q$ in $\text{Tm}(\diamond \triangleright D) A$, then $\text{lam}(\text{lam}(\text{lam}(q[p]))) \equiv \text{lam}(\text{lam}(\text{lam} q))$, hence $\text{refl} : \text{Tm} \diamond (\text{isPfProp} A)$.

Left to right: we have $t : \text{Tm} \diamond \left(\text{Id}_{\Pi D (A \Rightarrow A \Rightarrow A)} (\text{lam}(\text{lam}(\text{lam}(q[p]))) (\text{lam}(\text{lam}(\text{lam} q)))) \right)$. Canonicity for Id implies that $\text{lam}(\text{lam}(\text{lam}(q[p]))) \equiv \text{lam}(\text{lam}(\text{lam} q))$, hence

$$\text{app}(\text{app}(\text{app}(\text{lam}(\text{lam}(\text{lam}(q[p])))))) \equiv \text{app}(\text{app}(\text{app}(\text{lam}(\text{lam}(\text{lam} q)))))$$

Now using $\Pi\beta$ three times on both sides we obtain $q[p] \equiv q$ where both sides are in $\text{Tm}(\diamond \triangleright D \triangleright A \triangleright A[p]) (A[p][p])$, and this is $\text{isExtPfProp} A$. ◀

► **Corollary 11.** *In a model of type theory with Π , Id and canonicity, given a closed type A , $\text{Tm} \diamond (\text{isPfProp} A)$ if and only if $\text{isExtPfProp} A$.*

In an open context, external point-free propositions are stronger than internal ones.

► **Proposition 12.** *In a model of type theory with Π , Id , a type U and a family over it El (a possibly empty universe) and normalisation, we have $A : \text{Ty} \Gamma$ such that $\text{Tm} \Gamma (\text{isPfProp} A)$, but not $\text{isExtPfProp} A$.*

Proof. Pick $\Gamma := \diamond \triangleright U \triangleright \text{Id}_{\text{El} q \Rightarrow \text{El} q \Rightarrow \text{El} q} (\text{lam}(\text{lam}(q[p]))) (\text{lam}(\text{lam} q))$ and $A := \text{El}(q[p])$. Now $q[p]$ and q both in $\text{Tm}(\Gamma \triangleright A \triangleright A[p]) (A[p] \circ p)$ have different normal forms. ◀

Next, we describe the relationship of homotopy and point-free propositions. Here we use the non-dependent variants.

► **Proposition 13.**

- (i) *In a model of type theory with Π and Id , $\text{isPfProp} A$ implies $\text{isHProp} A$.*
- (ii) *In a model of type theory with Π , Id , an inductively defined \perp and normalisation,*
 - (a) *we have $\text{isHProp} \perp$, but not $\text{isPfProp} \perp$.*
 - (b) *we don't have that for any type A , $\text{isPfProp} (\text{isPfProp} A)$.*
- (iii) *In a model of type theory with Π , Id and function extensionality, $\text{isHProp} A$ implies $\text{isPfProp} A$.*

Proof.

- (i) We work internally. Given $p_A : \text{isPfProp} A \equiv (\lambda(a a' : A).a) = (\lambda a a'.a')$, we define $\lambda a a'.\text{ap}(\lambda z.z a a') p_A : \text{isHProp} A$.
- (ii) (a) Internally, we have $\lambda b.\text{elim}_\perp b : \text{isHProp} \perp$. Let's assume $\text{Tm} \diamond (\text{isPfProp} \perp)$. From Corollary 11 and Proposition 7, any two elements of \perp in any context are equal. But from normalisation we have $q[p] \not\equiv q : \text{Tm}(\diamond \triangleright \perp \triangleright \perp) \perp$ as they have different normal forms.
- (b) Assuming $\text{isPfProp} (\text{isPfProp} \perp)$, we obtain

$$q[p] \equiv q : \text{Tm}(\diamond \triangleright \text{isPfProp} \perp \triangleright \text{isPfProp} \perp) (\text{isPfProp} \perp)$$

the same way as in (a), but they have different normal forms.

6:12 Internal Strict Propositions Using Point-Free Equations

(iii) We have to show that $\text{isHProp } A$ implies $\text{isPfProp } A$. We work internally by the following double application of function extensionality.

$$\begin{array}{lcl}
 \text{isHProp } A & & \\
 ((a \ a' : A) \rightarrow a = a') & \equiv & \\
 ((a : A)(a' : A) \rightarrow (\lambda a'.a) a' = (\lambda a'.a') a') & \equiv & \\
 ((a : A) \rightarrow (\lambda a'.a) = (\lambda a'.a')) & \rightarrow \text{(function extensionality)} & \\
 ((a : A) \rightarrow (\lambda a \ a'.a) a = (\lambda a \ a'.a') a) & \equiv & \\
 (\lambda a \ a'.a') = (\lambda a \ a'.a') & \rightarrow \text{(function extensionality)} & \\
 \text{isPfProp } A & \equiv & \blacktriangleleft
 \end{array}$$

From the previous section, we know that TyP can be defined using isExtPfProp . If we start with a model that already has TyP , it is natural to ask about the relationship of TyP and the other notions of being a proposition.

► Proposition 14.

- (i) In a model of type theory with Π , Id and TyP , if $A : \text{TyP } \Gamma$, then $\text{Tm } \Gamma (\text{isHProp } (\uparrow A))$, $\text{Tm } \Gamma (\text{isPfProp } (\uparrow A))$ and $\text{isExtPfProp } (\uparrow A)$.
- (ii) In a model of type theory with Π , Σ and Id , if for every type A , $\text{isHProp } A$ implies $\text{isExtPfProp } A$, then the model has equality reflection.

Proof.

- (i) Because any two terms of type $\uparrow A$ are definitionally equal by irr , internally $\lambda a \ a'.\text{refl} : \text{isHProp } A$ and $\text{refl} : \text{isPfProp } A$.
- (ii) The proof is from [13]. Singleton types are in hProp , that is, internally $\text{isHProp } ((a' : A) \times \text{Id}_A a \ a')$ holds for any a , but if $\text{isExtPfProp } ((a' : A) \times \text{Id}_A a \ a')$, then for any $e : \text{Id}_A a \ a'$, we have $(a, \text{refl}) \equiv (a', e)$, hence $a \equiv \pi_1 (a, \text{refl}) \equiv \pi_1 (a', e) \equiv a'$. ◀

6 The setoid model externally

In this section, from a model of type theory with \top , Σ and Π , we build another model of type theory with the same type formers and a strict identity type, a strong transport rule and function extensionality. Strictness of the identity type means that any two elements of the identity type are definitionally equal (it is an external point-free proposition, isExtPfProp). Strength of transport means that we can transport an element of any family of types, not only families of strict propositions. In contrast, Agda and the method described in [13] only support a strict identity type with a weak transport: the identity type is Prop -valued and we can only transport along Prop -valued families.

In Section 7, we describe an internal version of this model construction where we define a model internally to an intensional metatheory. Section 7 relates to this section as the section on internal point-free propositions (Section 3) relates to the section on external point-free propositions (Section 4). The model construction in this section follows those in [4, 3] with some small improvements, but is defined in a more restricted setting: we do not assume that the input model has a universe of strict propositions.

Note that even if our metatheory is extensional type theory, we do not rely on any extensionality features in the input model. We only use an extensional metatheory for convenience. Following Hofmann's conservativity theorem [14], our arguments can be replayed in an intensional metatheory with function extensionality and uniqueness of identity proofs.

► **Construction 15.** *From an input model of type theory with \top , Σ , Π , a sort TyP closed under $\top\text{P}$, ΣP and TyP (as in Figure 3), we construct a model of type theory with the same type formers and a TyP -valued identity type with a strong transport rule and function extensionality.*

Construction. A context in the output model is a context in the input model together with an hProp -valued equivalence relation on substitutions into that context. Note that as our metatheory is extensional type theory, hProp and pfProp coincide.

$$\begin{aligned} \text{Con} &::= (|\Gamma| \quad : \text{Con}) \\ &\times (\Gamma^\sim \quad : \text{Sub } \Xi \mid \Gamma \mid \rightarrow \text{Sub } \Xi \mid \Gamma \mid \rightarrow \text{hProp}) \\ &\times (-[-]_\Gamma : \Gamma^\sim \gamma_0 \gamma_1 \rightarrow (\xi : \text{Sub } \Xi' \Xi) \rightarrow \Gamma^\sim (\gamma_0 \circ \xi) (\gamma_1 \circ \xi)) \\ &\times (\mathbf{R}_\Gamma \quad : (\gamma : \text{Sub } \Xi \mid \Gamma \mid) \rightarrow \Gamma^\sim \gamma \gamma) \\ &\times (\mathbf{S}_\Gamma \quad : \Gamma^\sim \gamma_0 \gamma_1 \rightarrow \Gamma^\sim \gamma_1 \gamma_0) \\ &\times (\mathbf{T}_\Gamma \quad : \Gamma^\sim \gamma_0 \gamma_1 \rightarrow \Gamma^\sim \gamma_1 \gamma_2 \rightarrow \Gamma^\sim \gamma_0 \gamma_2) \end{aligned}$$

In [4], the relation for contexts was TyP -valued, not metatheoretic proposition (hProp)-valued. We chose to use hProp for reasons of modularity: now the category part of the output model (Con , Sub) only refers to the category part of the input model. Note that the relation for types is TyP -valued.

Substitutions are substitutions in the input model which respect the relation.

$$\text{Sub } \Delta \Gamma := (|\gamma| : \text{Sub } |\Delta| \mid \Gamma \mid) \times (\gamma^\sim : \Delta^\sim \delta_0 \delta_1 \rightarrow \Gamma^\sim (|\gamma| \circ \delta_0) (|\gamma| \circ \delta_1))$$

Composition and identities are composition and identities from the input model where the \sim components are defined by function composition and the identity function. In fact, up to Π types, all the $|-|$ components in the output model are the corresponding components of the input model.

The empty context is defined as $|\diamond| := \diamond$ and $\diamond^\sim \sigma_0 \sigma_1 := \top$ which is trivially an equivalence relation.

Types are displayed setoids with TyP -valued relations together with coercion and coherence operations.

$$\begin{aligned} \text{Ty } \Gamma &::= \\ &(|A| \quad : \text{Ty } \mid \Gamma \mid) \\ &\times (A^\sim \quad : \Gamma^\sim \gamma_0 \gamma_1 \rightarrow \mathbf{Tm } \Xi (|A|[\gamma_0]) \rightarrow \mathbf{Tm } \Xi (|A|[\gamma_1]) \rightarrow \text{TyP } \Xi) \\ &\times (A^\sim [] : (A^\sim \gamma_{01} a_0 a_1)[\xi] \equiv A^\sim (\gamma_{01}[\xi]_\Gamma) (a_0[\xi]) (a_1[\xi])) \\ &\times (\mathbf{R}_A \quad : (a : \mathbf{Tm } \Xi (|A|[\gamma])) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim (\mathbf{R}_\Gamma \gamma) a a)) \\ &\times (\mathbf{S}_A \quad : \mathbf{Tm } \Xi (\uparrow A^\sim \gamma_{01} a_0 a_1) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim (\mathbf{S}_\Gamma \gamma_{01}) a_1 a_0)) \\ &\times (\mathbf{T}_A \quad : \mathbf{Tm } \Xi (\uparrow A^\sim \gamma_{01} a_0 a_1) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim \gamma_{12} a_1 a_2) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim (\mathbf{T}_\Gamma \gamma_{01} \gamma_{12}) a_0 a_2)) \\ &\times (\text{coe}_A : \mathbf{Tm } \Xi (\uparrow \Gamma^\sim \gamma_0 \gamma_1) \rightarrow \mathbf{Tm } \Xi (|A|[\gamma_0]) \rightarrow \mathbf{Tm } \Xi (|A|[\gamma_1])) \\ &\times (\text{coe}_A [] : \text{coe}_A \gamma_{01} a_0[\xi] \equiv \text{coe}_A (\gamma_{01}[\xi]_\Gamma) (a_0[\xi])) \\ &\times (\text{coh}_A : (\gamma_{01} : \mathbf{Tm } \Xi (\uparrow \Gamma^\sim \gamma_0 \gamma_1))(a_0 : \mathbf{Tm } \Xi (|A|[\gamma_0])) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim \gamma_{01} a_0 (\text{coe}_A \gamma_{01} a_0))) \end{aligned}$$

Type substitution is given by type substitution in the input model and function composition for the other components.

Terms are terms which respect the (displayed) equivalence relations.

$$\mathbf{Tm } \Gamma A := (|t| : \mathbf{Tm } \mid \Gamma \mid |A|) \times (t^\sim : (\gamma_{01} : \Gamma^\sim \gamma_0 \gamma_1) \rightarrow \mathbf{Tm } \Xi (\uparrow A^\sim \gamma_{01} (|t|[\gamma_0]) (|t|[\gamma_1])))$$

Term substitution is given by term substitution in the input model and function composition for the \sim component.

6:14 Internal Strict Propositions Using Point-Free Equations

Context extension is context extension $|\Gamma \triangleright A| := |\Gamma| \triangleright |A|$, the relation is given by metatheoretic Σ types: $(\Gamma \triangleright A) \sim (\gamma_0, a_0) (\gamma_1, a_1) := (\gamma_{01} : \Gamma \sim \gamma_0 \gamma_1) \times \text{Tm} \Xi (\uparrow A \sim \gamma_{01} a_0 a_1)$. This is an equivalence relation because $\Gamma \sim$ is an equivalence relation and $A \sim$ is a displayed equivalence relation. The \sim components of $-$, $-$, \mathbf{p} and \mathbf{q} are given by pairing and projections for metatheoretic Σ types. The equations $\triangleright \beta_1, \triangleright \beta_2, \triangleright \eta$ follow from β, η for metatheoretic Σ types. The unit type \top is given by $|\top| := \top, \top \sim \gamma_{01} t_0 t_1 := \top \mathbf{P}$.

Σ types use $\Sigma \mathbf{P}$ for the relation: we define $|\Sigma A B| := \Sigma |A| |B|$ and

$$(\Sigma A B) \sim \gamma_{01} (a_0, b_0) (a_1, b_1) := \Sigma \mathbf{P} (A \sim \gamma_{01} a_0 a_1) (B \sim (\gamma_{01} [\mathbf{p}], \mathbf{q}) (b_0 [\mathbf{p}]) (b_1 [\mathbf{p}])).$$

All the other components are pointwise, for example $\mathbf{R}_{\Sigma A B} (a, b) := (\mathbf{R}_A a, \mathbf{P} \mathbf{R}_B b)$ and

$$\text{coe}_{\Sigma A B} \gamma_{01} (a_0, b_0) := (\text{coe}_A \gamma_{01} a_0, \text{coe}_B (\gamma_{01}, \text{coh}_A \gamma_{01} a_0) b_0).$$

Pairing, first and second projection and the computation rules are straightforward. Note that to prove e.g. $\pi_1 (a, b) \equiv a$, it is enough to compare the first components, i.e. $|\pi_1 (a, b)| \equiv |a|$ as the second components are equal by irr .

For Π types, the $|-|$ component includes \sim components of the constituent types:

$$\begin{aligned} |\Pi A B| &:= \\ &\Sigma (\Pi |A| |B|) \\ &\left(\Pi \mathbf{P} (|A| [\mathbf{p}]) \left(\Pi \mathbf{P} (|A| [\mathbf{p}^2]) \left(\Pi \mathbf{P} (\uparrow A \sim (\mathbf{R}_\Gamma \mathbf{p}^3) (\mathbf{q} [\mathbf{p}]) \mathbf{q}) \right. \right. \right. \\ &\quad \left. \left. \left. (B \sim (\mathbf{R}_\Gamma \mathbf{p}^4, \mathbf{q}) (\mathbf{q} [\mathbf{p}^3] \$ \mathbf{q} [\mathbf{p}^2]) (\mathbf{q} [\mathbf{p}^3] \$ \mathbf{q} [\mathbf{p}])) \right) \right) \right) \end{aligned}$$

Functions are given by functions which respect the relation: for any two elements of $|A|$ that are related by $A \sim$, the outputs of the function are related by $B \sim$. We wrote \mathbf{p}^2 for $\mathbf{p} \circ \mathbf{p}$. With variable names and without weakenings, the same definition is written

$$\Sigma (f : \Pi (a : |A|) . |B|) . \Pi \mathbf{P} (a_0 a_1 : |A|, a_{01} : \uparrow A \sim (\mathbf{R}_\Gamma \text{id}) a_0 a_1) . B \sim (\mathbf{R}_\Gamma \text{id}, a_{01}) (f \$ a_0) (f \$ a_1).$$

The relation for Π types says that two functions are related if they map related inputs to related outputs:

$$\begin{aligned} (\Pi A B) \sim \gamma_{01} t_0 t_1 &:= \\ \Pi \mathbf{P} (|A| [\gamma_0]) &\left(\Pi \mathbf{P} (|A| [\gamma_1 \circ \mathbf{p}]) \left(\Pi \mathbf{P} (\uparrow A \sim (\gamma_{01} [\mathbf{p}^2]_\Gamma) (\mathbf{q} [\mathbf{p}]) \mathbf{q}) \right. \right. \\ &\quad \left. \left. (B \sim (\gamma_{01} [\mathbf{p}^3]_\Gamma, \mathbf{q}) (t_0 [\mathbf{p}^3] \$ (\mathbf{q} [\mathbf{p}^2])) (t_1 [\mathbf{p}^3] \$ (\mathbf{q} [\mathbf{p}])) \right) \right) \end{aligned}$$

Reflexivity for Π types is second projection: $\mathbf{R}_{\Pi A B} t := \pi_2 t$. The other components are defined as in [4]. The definition of lam and app are straightforward. Just as for Π , the definition of $|\text{lam } t|$ involves both $|t|$ and $t \sim$. When comparing two elements of $|\Pi A B|$ for equality, only the first components of the Σ types have to be compared, the second components are equal by irr .

The sort TyP is defined by TyPs in the input model together with coercion.

$$\text{TyP } \Gamma := (|A| : \text{TyP } |\Gamma|) \times (\text{coe}_A : \Gamma \sim \gamma_0 \gamma_1 \rightarrow \text{Tm} \Xi (\uparrow |A| [\gamma_0]) \rightarrow \text{Tm} \Xi (\uparrow |A| [\gamma_1]))$$

Compared to Ty which had nine components, TyP has only two. All the other components that Ty had are irrelevant for propositional types. Lifting is given by lifting in the input model, the relation is trivial and coercion comes from the coercion component in TyP :

$$|\uparrow A| := \uparrow |A| \qquad (\uparrow A) \sim \gamma_{01} a_0 a_1 := \top \mathbf{P} \qquad \text{coe}_{\uparrow A} \gamma_{01} a_0 := \text{coe}_A \gamma_{01} a_0$$

TyP is closed under \top P, Σ P and Π P.

Thus we constructed a model of type theory with \top , Π , Σ and a sort TyP closed under the same type formers.

This model has an identity type $\text{Id}_A a a' : \text{TyP } \Gamma$ for $a, a' : \text{Tm } \Gamma A$.

$$\begin{array}{c} |\text{Id}_A a a'| := A \sim (\text{R}_\Gamma \text{id}) a a' \\ \text{coe}_{\text{Id}_A a a'} \gamma_{01} \quad \underbrace{e}_{:A \sim (\text{R}_\Gamma \gamma_0) (a[\gamma_0]) (a'[\gamma_0])} \quad := \text{T}_A \quad \underbrace{(\text{Id}_A a a') \sim \gamma_{01} e_0 e_1}_{:A \sim (\text{S}_\Gamma \gamma_{01}) (a[\gamma_1]) (a[\gamma_0])} \quad \underbrace{(\text{T}_A e \quad (a' \sim \gamma_{01}))}_{:A \sim \gamma_{01} (a'[\gamma_0]) (a'[\gamma_1])} \quad) \\ \underbrace{\hspace{15em}}_{:A \sim (\text{R}_\Gamma \gamma_1) (a[\gamma_1]) (a'[\gamma_1])} \end{array}$$

It has a constructor `refl` and an eliminator `transp` (J is a consequence of transport as equality is proof-irrelevant).

$$\begin{array}{ll} \text{refl} & : \text{Tm } \Gamma (\uparrow \text{Id}_A a a) \\ |\text{refl}| & := \text{R}_A a \\ \text{refl} \sim \gamma_{01} & := \text{ttP} \\ \text{transp} & : (P : \text{Ty} (\Gamma \triangleright A)) \rightarrow \text{Tm } \Gamma (\uparrow \text{Id}_A a a') \rightarrow \text{Tm } \Gamma (P[\text{id}, a]) \rightarrow \text{Tm } \Gamma (P[\text{id}, a']) \\ |\text{transp } P e u| & := \text{coe}_P (\text{R}_\Gamma \text{id}, |e|) |u| \end{array}$$

The computation rule of `transp` only holds up to `Id`, but as described in [4], the model can be refined to support a definitional computation rule. Note that transport works with arbitrary Ty-motive, the motive does not have to be TyP (as opposed to the inductively defined Prop-valued identity type in Agda). Function extensionality holds by definition of the identity type. \blacktriangleleft

► Construction 16. *From an input model of type theory with \top , Σ , Π , we construct a model of type theory with \top , Σ , Π , a sort of propositions TyP closed under \top , Σ , Π and a TyP-valued identity type with a strong transport rule and function extensionality.*

Construction. We take the input model, equip it with TyP using Construction 9, then invoke Construction 15. \blacktriangleleft

The above construction can be extended with the empty type: if the input model has $\perp : \text{Ty}$, the output model also supports $\perp : \text{Ty}$ with its elimination rule, but we do not have $\perp : \text{TyP}$ (unless `isExtPfProp \perp` in the input model). Similarly, to justify booleans in the output model, we need that the input model has booleans and a definitionally proof-irrelevant family over booleans that we can use to define identity for booleans:

$$\begin{array}{ll} \text{IdBool} & : \text{Ty} (\Gamma \triangleright \text{Bool} \triangleright \text{Bool}) \\ \text{Idtrue} & : \text{Tm } \Gamma (\text{IdBool}[\text{id}, \text{true}, \text{true}]) \\ \text{Idfalse} & : \text{Tm } \Gamma (\text{IdBool}[\text{id}, \text{false}, \text{false}]) \\ \text{Idirr} & : (e e' : \text{Tm } (\Gamma \triangleright \text{Bool} \triangleright \text{Bool}) \text{IdBool}) \rightarrow e \equiv e' \end{array}$$

But then we might as well require TyP in the input model with closure under inductive types.

7 The setoid model internally

In the previous section we showed how a setoid model can be constructed without requiring a sort `TyP` in the input model. Can we redo the same internally to intensional type theory using point-free propositions? That is, can we define a setoid model in Agda (which can be viewed as the initial model of intensional type theory) without using strict propositions (`Prop`, `TyP`)?

Compared to Construction 15 of the previous section, the role of the input model is taken by our metatheory (Agda), the role of the output model is the model we construct. The equations of our model are given by the identity type of the metatheory. If all the equations can be proven by `refl`, it means that the model is strict. In such a case an external model construction can be obtained from the internal model (see [4, Section 3] for an exposition of model constructions vs. internal models through the example of the graph model). Model constructions are also called syntactic translations, see [7] for such a presentation.

The notion of model we construct is described in Figures 1, 2, 3 in extensional type theory. As some operations and equations typecheck only because of previous equations (e.g. `lam[]` depends on `Π[]`), the complete intensional description of the notion of model has many transports compared to this (see [5] for an exposition using explicit transports). However if an equation is proved by `refl` in the model, then transports over it disappear, so *concrete* strict models can be defined in Agda without using any transports.

External model constructions where the definitions of types (and substitutions and terms) don't involve equations can be internalised immediately as strict models. This is the case for the setoid model using `TyP`, see [4]. In our case however, there is an equation expressing that the equivalence relation is a proposition. This makes the construction more involved as we have to prove that the witnesses of propositionality are equal.

The answer to the above question is yes. This section was formalised in Agda [12].

► **Construction 17.** *We construct a model of type theory with \perp , \top , Σ , Π , a sort of propositions `TyP` closed under \top , Σ , Π , a `TyP`-valued identity type with a strong transport rule and function extensionality. All equations of our model hold definitionally, with the exceptions `irr`, `Σ[]`, `,[]`, `Πη`, `Π[]`, `lam[]`.*

Construction. We explain the main components, for details consult the formalisation.

We define contexts as setoids where the equivalence relation is a point-free proposition. Compare it with how contexts were defined in the external Construction 15.

$$\begin{aligned} \text{Con} &::= (|\Gamma| : \text{Type}) \\ &\times (\Gamma^\sim : |\Gamma| \times |\Gamma| \rightarrow \text{Type}) \\ &\times (\Gamma^{\text{P}} : \text{isPfpPropd } \Gamma^\sim) \\ &\times (\mathbf{R}_\Gamma : (\gamma_x : |\Gamma|) \rightarrow \Gamma^\sim (\gamma_x, \gamma_x)) \\ &\times (\mathbf{S}_\Gamma : \Gamma^\sim (\gamma_0, \gamma_1) \rightarrow \Gamma^\sim (\gamma_1, \gamma_0)) \\ &\times (\mathbf{T}_\Gamma : \Gamma^\sim (\gamma_0, \gamma_1) \rightarrow \Gamma^\sim (\gamma_1, \gamma_2) \rightarrow \Gamma^\sim (\gamma_0, \gamma_2)) \end{aligned}$$

We don't have equations on contexts, so it is not an issue that there is an equation (Γ^{P}) as one of the components. There will be an issue for types, see below.

Substitutions are functions that respect the relations.

$$\text{Sub } \Delta \Gamma ::= (|\gamma| : |\Delta| \rightarrow |\Gamma|) \times (\gamma^\sim : \Delta^\sim (\delta_0, \delta_1) \rightarrow \Gamma^\sim (|\gamma| \delta_0, |\gamma| \delta_1))$$

They form a category with function composition (for both $|-|$ and $- \sim$ components) and the identity function. The categorical laws are definitional. The empty context is given by \top with the constant \top relation.

Types are displayed setoids with coercion and coherence (note that later we will replace types by their strictified variants).

$$\begin{aligned}
\text{Ty } \Gamma &::= \\
&(|A| \quad : |\Gamma| \rightarrow \text{Type}) \\
&\times (A^\sim \quad : (\gamma_0 : |\Gamma|) \times (\gamma_1 : |\Gamma|) \times \Gamma^\sim \gamma_0 \gamma_1 \times |A| \gamma_0 \times |A| \gamma_1 \rightarrow \text{Type}) \\
&\times (A^P \quad : \text{isPfPropd } A^\sim) \\
&\times (R_A \quad : (a_x : |A| \gamma_x) \rightarrow A^\sim (\gamma_x, \gamma_x, R_\Gamma \gamma_x, a_x, a_x)) \\
&\times (S_A \quad : A^\sim (\gamma_0, \gamma_1, \gamma_{01}, a_0, a_1) \rightarrow A^\sim (\gamma_1, \gamma_0, S_\Gamma \gamma_{01}, a_1, a_0)) \\
&\times (T_A \quad : A^\sim (\gamma_0, \gamma_1, \gamma_{01}, a_0, a_1) \rightarrow A^\sim (\gamma_1, \gamma_2, \gamma_{12}, a_1, a_2) \rightarrow A^\sim (\gamma_0, \gamma_2, T_\Gamma \gamma_{01} \gamma_{12}, a_0, a_2)) \\
&\times (\text{coe}_A : \Gamma^\sim (\gamma_0, \gamma_1) \rightarrow |A| \gamma_0 \rightarrow |A| \gamma_1) \\
&\times (\text{coh}_A : (\gamma_{01} : \Gamma^\sim (\gamma_0, \gamma_1))(a_0 : |A| \gamma_0) \rightarrow A^\sim (\gamma_0, \gamma_1, \gamma_{01}, a_0, \text{coe}_A \gamma_{01} a_0))
\end{aligned}$$

Compared to the external version, we don't need substitution laws ($A^\sim []$ and $\text{coe}_A []$) and instead of making the relation Prop -valued we add an element of the identity type saying that A^\sim is a point-free proposition. We can prove that two types are equal if their $|-|$, $- \sim$, $-^P$, coe components are equal. The other components will be equal by $-^P$. Unfortunately, due to Proposition 13 part (ii) (b), we have to show that the proofs of propositionality $-^P$ coincide.

Substitution of types is given by function composition for the $|-|$ and $- \sim$ components, for the $-^P$ component we use the fact that dependent point-free propositions are closed under reindexing. The reflexivity, symmetry and transitivity components of $A[\gamma]$ are constructed using transport and the corresponding components of A . The exact way they are constructed does not matter as they are proof irrelevant by A^P . We prove the substitution laws $[o]$ and $[\text{id}]$ up to the identity type using J .

Terms are like substitutions, but with dependent functions.

$$\text{Tm } \Gamma A ::= (|t| : (\gamma_x : |\Gamma|) \rightarrow |A| \gamma_x) \times (t^\sim : (\gamma_{01} : \Gamma^\sim (\gamma_0, \gamma_1)) \rightarrow A^\sim (\gamma_0, \gamma_1, \gamma_{01}, |t| \gamma_0, |t| \gamma_1))$$

The $|-|$ and $- \sim$ components of context extension are given by Σ types, the propositionality component is using the fact that point-free propositions are closed under Σ .

Analogously to the model in the previous section, we can show that we have \perp , \top , Σ and Π types. The β rules are definitional for both Σ and Π , however for Π the η rule only holds up to the metatheoretic identity type. The reason is that $|\Pi A B|$ is defined as a Σ type consisting of a function from $|A|$ to $|B|$ and a proof that it respects the relation.

$$\begin{aligned}
|\Pi A B| \gamma_x &::= (f : (a_x : |A| \gamma_x) \rightarrow |B| (\gamma_x, a_x)) \times \\
&(a_{01} : A^\sim (\gamma_x, \gamma_x, R_\Gamma \gamma_x, a_0, a_1)) \rightarrow B^\sim ((\gamma_x, a_0), (\gamma_x, a_1), (R_\Gamma \gamma_x, a_{01}), f a_0, f a_1)
\end{aligned}$$

Two functions are related by $(\Pi A B)^\sim$ if they map related inputs to related outputs. Hence there are two (definitionally) different ways of proving that a $t : \text{Tm } \Gamma (\Pi A B)$ respects a (homogeneous) relation $a_{01} : A^\sim (\gamma_x, \gamma_x, R_\Gamma \gamma_x, a_0, a_1)$. One is $\pi_2 (|t| \gamma_x) a_{01}$, the other is $t^\sim (R_\Gamma \gamma_x) a_{01}$. Because B^\sim is a proposition, these are equal, but only up to the identity type. And the eta rule computes to the usage of the two different versions on the two sides of the equation. We do not prove the substitution laws $\perp [], \top [], \Sigma [], \cdot [], \Pi [], \text{lam} []$ yet. There is no need to worry, we will prove them after replacing Ty with its strictified variant.

6:18 Internal Strict Propositions Using Point-Free Equations

If an equation is not definitional and there are later components in the model that depend on it (as $\text{lam}[]$ depends on $\Pi[]$), it makes the model construction extremely tedious. The situation one ends up in is also known as “transport hell”. As the functor laws $[\circ]$, $[\text{id}]$ for types and terms are not definitional, almost every operation that mentions substitutions involves transports. Instead of fighting in transport hell and proving the transported versions of the laws $\perp[]$, \dots , $\text{lam}[]$, we follow the local universes approach [20]. We wrap Ty into Ty' which contains a base context, a substitution into this context and a Ty in this base context.

$$\text{Ty}'\Gamma \equiv (\text{con}_A : \text{Con}) \times (\text{sub}_A : \text{Sub}\Gamma \text{con}_A) \times (\text{ty}_A : \text{Ty}\text{con}_A)$$

Substitution for Ty' is defined as composition in the sub component, and as composition in the category is definitional, the laws $[\circ]$, $[\text{id}]$ become definitional. Terms Tm' and context extension $-\triangleright'$ can be defined, and all the CwF equations are definitional. The type formers can be redefined as their primed versions \perp' , Σ' and Π' . $\perp'[]$ and $\top'[]$ hold definitionally, but $\Sigma'[]$ and $\Pi'[]$ rely on definitional β and η for Σ and Π (the ones defined for Ty), and we are missing an η for Π . Hence $\Sigma[]$, $\cdot[]$, $\Pi\eta$, $\Pi[]$, $\text{lam}[]$ only hold up to the identity type.

We define $\text{TyP}\Gamma$ as those families over $|\Gamma|$ that are (point-free) propositional and which have coercion.

$$\text{TyP}\Gamma \equiv (|A| : |\Gamma| \rightarrow \text{Type}) \times (A^P : \text{isPfPropd } |A|) \times (\text{coe}_A : \Gamma \sim (\gamma_0, \gamma_1) \rightarrow |A| \gamma_0 \rightarrow |A| \gamma_1)$$

\uparrow is given by letting the relation be constant \top , and showing closure under \top , Σ and Π is straightforward. Proof irrelevance irr comes from the assumed equation A^P , hence it is not definitional. Definition of the TyP -valued identity type is analogous to the construction in the previous section. Strictification of TyP is analogous to that of Ty . \blacktriangleleft

We conjecture that without strictification (the replacement of Ty by Ty') we can still prove all the equations, however this seems to be very difficult due to “transport hell”.

8 Examples of strict algebraic structures

Point-free equations can be used to define strict variants of algebraic structures. For example, internally to a model of type theory with a universe Type closed under Π , Σ , Id , a strict monoid is defined as follows.

$$\begin{aligned} \mathbf{M} & : \text{Type} \\ - \otimes - & : \mathbf{M} \rightarrow \mathbf{M} \rightarrow \mathbf{M} \\ \text{ass} & : \text{Id}_{\mathbf{M} \rightarrow \mathbf{M} \rightarrow \mathbf{M} \rightarrow \mathbf{M}} (\lambda x y z. (x \otimes y) \otimes z) (\lambda x y z. x \otimes (y \otimes z)) \\ \mathbf{o} & : \mathbf{M} \\ \text{idl} & : \text{Id}_{\mathbf{M} \rightarrow \mathbf{M}} (\lambda x. \mathbf{o} \otimes x) (\lambda x. x) \\ \text{idr} & : \text{Id}_{\mathbf{M} \rightarrow \mathbf{M}} (\lambda x. x \otimes \mathbf{o}) (\lambda x. x) \end{aligned}$$

Compare it with the usual definition of monoid where the laws are stated using universal quantification:

$$\begin{aligned} \text{ass} & : (x y z : \mathbf{M}) \rightarrow \text{Id}_{\mathbf{M}} ((x \otimes y) \otimes z) (x \otimes (y \otimes z)) \\ \text{idl} & : (x : \mathbf{M}) \rightarrow \text{Id}_{\mathbf{M}} (\mathbf{o} \otimes x) x \\ \text{idr} & : (x : \mathbf{M}) \rightarrow \text{Id}_{\mathbf{M}} (x \otimes \mathbf{o}) x \end{aligned}$$

If our model has canonicity, then in the empty context, for any strict monoid, the laws hold definitionally. For example, booleans where conjunction is defined as $a \wedge b := \text{if } a \text{ then } b \text{ else false}$ do not form a strict monoid. We do have $\text{idl} : \text{true} \wedge b \equiv b$, but we don't have idr or associativity definitionally, only propositionally. So booleans with $- \wedge -$ form a usual monoid, but not a strict monoid. Similarly, natural numbers with addition form a usual monoid, but not a strict monoid.

In contrast, for any type A , the function space $A \rightarrow A$ forms a monoid with $f \otimes g := \lambda x. f (g x)$ and $\circ := \lambda x. x$. We have associativity as $\lambda f g h. (f \otimes g) \otimes h \equiv \lambda f g h x. f (g (h x)) \equiv \lambda f g h. f \otimes (g \otimes h)$ and the identity laws hold as e.g. $\lambda f. \circ \otimes f \equiv \lambda f x. f x \equiv \lambda f. f$.

Strict monoids are closed under finite products following the η rule for \times . We can define displayed strict monoids over a strict monoid, and dependent product of strict monoids. Strict monoids are also closed by A -ary products for any type A . That is, given a strict monoid with carrier M , $A \rightarrow M$ is also a strict monoid.

Point-free propositions are another strict algebraic structure with no operations and only one equation: any two elements are equal. Closure under \top and Σ give closure under (dependent) finite products, closure under Π is the same as having A -ary (dependent) products for any type A .

We conjecture that for any (generalised) algebraic structure, we have a CwF with \top , Σ and extensional Id of strict algebras internally to any model of intensional type theory. The category part of the CwF is the category of algebras and homomorphisms, terms and types are displayed algebras and sections, context extension is dependent product of algebras, and so on. This semantics was called finite limit CwF in [17].

The term “strict” algebraic structure is only correct in intensional type theory. In a model with function extensionality, strict and usual monoids coincide.

There is a stronger sense in which algebraic structures can be “strict”. Obviously, to define a strict monoid in the empty context, all laws have to hold definitionally. However when *assuming* a strict monoid and using it in a construction in this open context, the laws only hold up to propositional equality. It would be convenient to have implementations of type theory with strict algebraic structures in this stronger sense. Currently, Agda only supports one algebraic structure which is strict in this stronger sense: propositions.

9 Summary

In this paper we attempted to push the limits of what can be done in intensional type theory without function extensionality or uniqueness of identity proofs. We exploited the fact that in intensional type theory, in the empty context propositional and definitional equality coincide. We used this to define a dynamic universe of strict propositions internally. We expect that other strict algebraic structures with the expected properties can be defined along the same lines. In a strict algebraic structure, all equations are definitional. As we cannot assume definitional equalities in type theory, when we assume a member of a strict algebraic structure, the equations only hold propositionally. This makes it difficult to use such algebraic structures in practice. However we think that model constructions of type theory can be formalised as functions between strict models. We conjecture that the canonicity and normalisation displayed models from the corresponding proofs for type theory [10, 6] can be formalised in pure intensional type theory. These would be displayed over a strict model defined as a point-free algebraic structure. There are other inherent limitations of point-free propositions, e.g. the fact that we cannot prove that being a point-free proposition is a point-free proposition.

Internal strict models can be externalised directly. We would like to understand in which circumstances internal non-strict models can be externalised into model constructions. Another open problem is whether isHProp ($\text{isPfProp } A$) is provable in intensional type theory.

A strict proposition-valued identity type with a strong transport rule was used to define presheaves [22] and a universe of setoids closed under dependent function space [3]. It is not clear whether such a type theory has normalisation [1]. Currently the only justification that we know for this strong transport rule is the setoid model construction. We showed that such an identity type can be derived in intensional type theory using point-free propositions. It seems that our construction is limited, the model we constructed does not include a universe of propositions or inductive types. In the future, we would like to circumscribe the exact conditions that the input model has to satisfy in order to obtain inductive types and universes from the setoid model construction.

References

- 1 Andreas Abel and Thierry Coquand. Failure of normalization in impredicative type theory with proof-irrelevant propositional equality. *Log. Methods Comput. Sci.*, 16(2), 2020. doi:10.23638/LMCS-16(2:14)2020.
- 2 Thorsten Altenkirch. Extensional equality in intensional type theory. In *14th Annual IEEE Symposium on Logic in Computer Science, Trento, Italy, July 2-5, 1999*, pages 412–420. IEEE Computer Society, 1999. doi:10.1109/LICS.1999.782636.
- 3 Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, Christian Sattler, and Filippo Sestini. Constructing a universe for the setoid model. In Stefan Kiefer and Christine Tasson, editors, *Foundations of Software Science and Computation Structures - 24th International Conference, FOSSACS 2021, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2021, Luxembourg City, Luxembourg, March 27 - April 1, 2021, Proceedings*, volume 12650 of *Lecture Notes in Computer Science*, pages 1–21. Springer, 2021. doi:10.1007/978-3-030-71995-1_1.
- 4 Thorsten Altenkirch, Simon Boulier, Ambrus Kaposi, and Nicolas Tabareau. Setoid type theory—a syntactic translation. In Graham Hutton, editor, *Mathematics of Program Construction*, pages 155–196, Cham, 2019. Springer International Publishing.
- 5 Thorsten Altenkirch and Ambrus Kaposi. Type theory in type theory using quotient inductive types. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '16, pages 18–29, New York, NY, USA, 2016. ACM. doi:10.1145/2837614.2837638.
- 6 Thorsten Altenkirch and Ambrus Kaposi. Normalisation by Evaluation for Type Theory, in Type Theory. *Logical Methods in Computer Science*, Volume 13, Issue 4, October 2017. doi:10.23638/LMCS-13(4:1)2017.
- 7 Simon Boulier, Pierre-Marie Pédrot, and Nicolas Tabareau. The next 700 syntactical models of type theory. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 182–194, New York, NY, USA, 2017. ACM. doi:10.1145/3018610.3018620.
- 8 Simon Castellan, Pierre Clairambault, and Peter Dybjer. Categories with families: Untyped, simply typed, and dependently typed. *CoRR*, abs/1904.00827, 2019. arXiv:1904.00827.
- 9 Thierry Coquand. About the setoid model, 2013. URL: <http://www.cse.chalmers.se/~coquand/setoid.pdf>.
- 10 Thierry Coquand. Canonicity and normalization for dependent type theory. *Theor. Comput. Sci.*, 777:184–191, 2019. doi:10.1016/j.tcs.2019.01.015.
- 11 Thierry Coquand. Reduction free normalisation for a proof irrelevant type of propositions. *CoRR*, abs/2103.04287, 2021. arXiv:2103.04287.
- 12 István Donkó and Ambrus Kaposi. Agda formalization for the paper “Internal strict propositions using point-free equations”. <https://bitbucket.org/akaposi/prop>, 2022.

- 13 Gaëtan Gilbert, Jesper Cockx, Matthieu Sozeau, and Nicolas Tabareau. Definitional proof-irrelevance without K. *Proc. ACM Program. Lang.*, 3(POPL):3:1–3:28, 2019. doi:10.1145/3290316.
- 14 Martin Hofmann. Conservativity of equality reflection over intensional type theory. In *TYPES 95*, pages 153–164, 1995.
- 15 Martin Hofmann. *Extensional constructs in intensional type theory*. CPHC/BCS distinguished dissertations. Springer, 1997.
- 16 Jasper Hugunin. Why not w? 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 8:1–8:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.TYPES.2020.8.
- 17 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, January 2019. doi:10.1145/3290315.
- 18 Ambrus Kaposi and Zongpu Xie. Quotient inductive-inductive types in the setoid model. In Henning Basold, editor, *27th International Conference on Types for Proofs and Programs, TYPES 2021*. Universiteit Leiden, 2021. URL: <https://types21.liacs.nl/download/quotient-inductive-inductive-types-in-the-setoid-model/>.
- 19 Nicolai Kraus and Jakob von Raumer. Coherence via well-foundedness: Taming set-quotients in homotopy type theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 662–675. ACM, 2020. doi:10.1145/3373718.3394800.
- 20 Peter Lefanu Lumsdaine and Michael A. Warren. The local universes model: An overlooked coherence construction for dependent type theories. *ACM Trans. Comput. Logic*, 16(3), July 2015. doi:10.1145/2754931.
- 21 Erik Palmgren. From type theory to setoids and back. *arXiv e-prints*, page arXiv:1909.01414, September 2019. arXiv:1909.01414.
- 22 Pierre-Marie Pédrot. Russian constructivism in a prefascist theory. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8-11, 2020*, pages 782–794. ACM, 2020. doi:10.1145/3373718.3394740.
- 23 The Univalent Foundations Program. *Homotopy type theory: Univalent foundations of mathematics*. Technical report, Institute for Advanced Study, 2013.

Constructive Cut Elimination in Geometric Logic

Giulio Fellin   

Department of Computer Science, University of Verona, Italy

Department of Mathematics, University of Trento, Italy

Department of Philosophy, History and Art Studies, University of Helsinki, Finland

Sara Negri   

Department of Mathematics, University of Genoa, Italy

Eugenio Orlandelli   

Department of Philosophy and Communication Studies, University of Bologna, Italy

Abstract

A constructivisation of the cut-elimination proof for sequent calculi for classical and intuitionistic infinitary logic with geometric rules – given in earlier work by the second author – is presented. This is achieved through a procedure in which the non-constructive transfinite induction on the commutative sum of ordinals is replaced by two instances of Brouwer’s Bar Induction. Additionally, a proof of Barr’s Theorem for geometric theories that uses only constructively acceptable proof-theoretical tools is obtained.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases Geometric theories, sequent calculi, axioms-as-rules, infinitary logic, constructive cut elimination

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.7

Funding Partially funded by the Academy of Finland, research project no. 1308664.

Acknowledgements We are very grateful to Peter Schuster for precious comments and helpful discussions on various points. We also thank the three anonymous referees for their detailed and insightful comments that helped improving the paper.

1 Introduction

Notable parts of algebra and geometry can be formalised as *coherent theories* over first-order classical or intuitionistic logic. Their axioms are *coherent implications*, i.e., universal closures of implications $D_1 \supset D_2$, where both D_1 and D_2 are built up from atoms using conjunction, disjunction and existential quantification. Examples include all algebraic theories, such as the theory of groups and the theory of rings, all essentially algebraic theories, such as category theory [7], the theory of fields, the theory of local rings, lattice theory [22], projective and affine geometry [22, 17], the theory of separably closed local rings (aka “strictly Henselian local rings”) [9, 17, 25].

Although wide, the class of coherent theories leaves out certain axioms used in algebra such as the axioms of torsion Abelian groups or of Archimedean ordered fields, or used in the theory of connected graphs, as well as in the modelling of epistemic social notions such as common knowledge. All the latter examples can however be axiomatised by means of *geometric axioms*, a generalization of coherent axioms that allows infinitary disjunctions.

Geometric implications give a Glivenko class [18], as shown by Barr’s Theorem:

► **Theorem 1** (Barr’s Theorem [3]). *If \mathcal{T} is a coherent (geometric) theory and A is a coherent (geometric) sentence provable from \mathcal{T} in (infinitary) classical logic, then A is provable from \mathcal{T} in (infinitary) intuitionistic logic.*



© Giulio Fellin, Sara Negri, and Eugenio Orlandelli;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 7; pp. 7:1–7:16



Leibniz International Proceedings in Informatics

Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Barr’s Theorem¹ has its origin, through appropriate completeness results, in the theory of sheaf models, with the following formulation:

► **Theorem 2** ([12], Ch.9, Thm.2). *For every Grothendieck topos \mathcal{E} there exists a complete Boolean algebra \mathbf{B} and a surjective geometric morphism $Sh(\mathbf{B}) \rightarrow \mathcal{E}$.*

An extremely simple and purely syntactic proof of the first-order Barr’s Theorem for coherent theories has been given in [14] by means of **G3** sequent calculi: it is shown how to express coherent implications by means of rules that preserve admissibility of the structural rules. As a consequence, Barr’s theorem is proved by simply noticing that a proof in **G3C.G** – i.e., the calculus for classical logic extended with rules for coherent implications – is also a proof in the intuitionistic multisuccedent calculus **G3I.G**.

In [16], this approach to Barr’s Theorem has been generalized to (infinitary) geometric theories using **G3**-style calculi for classical and intuitionistic infinitary logic **G3[CI] $_{\omega}$** (with finite sequents instead of countably infinite sequents) and their extension with rules expressing geometric implications **G3[CI] $_{\omega}$.G**. To illustrate, the geometric axiom of torsion Abelian groups

$$\forall x. \bigvee_{n>0} nx = 0$$

is expressed by the infinitary rule

$$\frac{\{nx = 0, \Gamma \Rightarrow \Delta \mid n > 0\}}{\Gamma \Rightarrow \Delta} .$$

The main results in [16] are that in **G3[CI] $_{\omega}$.G** all rules are height-preserving invertible, the structural rules of weakening and contraction are height-preserving admissible, and cut is admissible. Hence, Barr’s Theorem for geometric theories is proved in [16] as it was done in [14] for coherent ones: a proof in **G3C $_{\omega}$.G** is also a proof in the intuitionistic multisuccedent calculus **G3I $_{\omega}$.G**.

We observe that the cut-elimination procedure given in Sect. 4.1 of [16] is not constructive. This is an instance of a typical limitation of cut eliminations in infinitary logics [6, 11, 23] since these proofs use the “natural” (or Hessenberg) commutative sum of ordinals $\alpha\#\beta$:

$$(\omega^{\alpha_m} + \dots + \omega^{\alpha_0}) \# (\omega^{\beta_n} + \dots + \omega^{\beta_0}) = (\omega^{\gamma_{m+n+1}} + \dots + \omega^{\gamma_0})$$

where $\gamma_{m+n+1}, \dots, \gamma_0$ is a decreasing permutation of $\alpha_m, \dots, \alpha_0, \beta_n, \dots, \beta_0$; see [24, 10.1.2B]. The resort to the natural sum is inescapable for proofs using cut-height (i.e., the sum of the derivation-height of the premisses of cut) as inductive parameter: it ensures that we can apply the inductive hypothesis when permuting the cut upwards in the derivation of one premisses. Nevertheless, it makes the proof non-constructive since

[its] definition utilises the Cantor normal form of ordinals to base ω . This normal form is not available in **CZF** [Constructive Zermelo–Fraenkel set theory] (or **IZF** [Intuitionistic Zermelo–Fraenkel set theory]) and thus a different approach is called for. [20, p. 369]

¹ Barr’s theorem is often alleged to achieve more in that it also allows to eliminate uses of the axiom of choice. That such formulations of Barr’s theorem should be taken with caution is demonstrated in [20] where the *internal* vs. *external* addition of the the axiom of choice is considered and it is shown that the latter preserves conservativity whereas the former does not.

We constructivise² the cut-elimination proof for $\mathbf{G3[CI]_{\omega}.G}$ by giving a procedure that replaces the induction on the cut-height with two transfinite inductions on the height of the derivations of the right and the left premiss of cut respectively – see Lemmas 20 and 21 – and it replaces the main induction on the depth of the cut-formula with two instances of Brouwer’s principle of Bar Induction – see Theorem 23.³ As a consequence, we are able to give a proof of Barr’s Theorem for geometric theories that uses only constructively acceptable proof-theoretic tools. Moreover, our proof strategy allows to constructivise the cut-elimination procedure for other infinitary calculi.

2 Syntax and sequent calculi for infinitary logics

Let \mathcal{S} be a signature containing, for every $n \in \mathbb{N}$, a countable (i.e., finite, possibly empty, or countably infinite) set $REL_n^{\mathcal{S}}$ of n -ary predicate letters P_1^n, P_2^n, \dots , and a countable set CON of individual constants c_1, c_2, \dots . Let VAR be a denumerable set of variables x_1, x_2, \dots . The language contains the following logical symbols: $=, \top, \perp, \wedge, \vee, \supset, \forall, \exists$, as well as countable conjunction $\bigwedge_{n>0}$ and countable disjunction $\bigvee_{n>0}$.

The sets TER of *terms* is the union of VAR and CON . The set of *formulas* of the language $\mathcal{L}_{\omega}^{\mathcal{S}}$ is generated by:

$$A ::= P_i^n t_1, \dots, t_n \mid t_1 = t_2 \mid \top \mid \perp \mid A \wedge A \mid A \vee A \mid A \supset A \mid \forall x A \mid \exists x A \mid \bigwedge_{n>0} A_n \mid \bigvee_{n>0} A_n$$

where $t_i \in TER$, $P_i^n \in REL_n^{\mathcal{S}}$, and $x \in VAR$.

We use the following metavariables:

- x, y, z for variables and $\vec{x}, \vec{y}, \vec{z}$ for lists thereof;
- t, s, r for terms;
- P, Q, R for atomic formulas;
- A, B, C for formulas.

We use $A(\vec{x})$ to say that the variables having free occurrences in A are included in \vec{x} . We follow the standard conventions for parentheses. The formulas \top , $\neg A$ and $A \supset C B$ are defined as expected. When considering (infinitary) classical logic we can shrink the set of primitive logical symbols by means of the well-known De Morgan’s dualities (including $\bigvee_{n>0} A_n \supset C \neg \bigwedge_{n>0} \neg A$), however also in the classical case we consider a language where all operators (excluding \neg and $\supset C$) are taken as primitive. This is not just useful but even necessary since our purpose is to extract the constructive content of classical proofs and many of the interdefinabilities do not hold in intuitionistic logic.

The notions of *free* and *bound occurrences* of a variable in a formula are the usual ones. We posit that no formula may have infinitely many free variables. A *sentence* is a formula without free occurrences of variables. Given a formula A , we use $A(t/x)$ to denote the formula obtained by replacing each free occurrence of x in A with an occurrence of t , provided that t is free for x in A – i.e., no new occurrence of t is bound by a quantifier.

Each formula A has a countable ordinal $d(A)$ as its *depth* (the successor of the supremum of the depths of its immediate subformulas). More precisely

² By “constructive” here we mean not relying on classical logical principles such as excluded middle or linearity of ordinals but we do not mean acceptable in all schools of constructive mathematics.

³ See [20, §7] for a different proof, based on constructive ordinals, of cut elimination in infinitary logic. The proof in [2] does not use ordinals, but it is inherently classical in that it uses a one-sided calculus based on De Morgan’s dualities.

7:4 Constructive Cut Elimination in Geometric Logic

► **Definition 3** (Depth of A). $d(A) = \sup\{d(B) \mid B \text{ is an immediate subformula of } A\} + 1$.

For example, \perp and atoms P have depth 1, since they have no immediate subformulas and the supremum of an empty family of ordinals is 0. The definition of depth implies that, if A' is a proper subformula of A , then $d(A') < d(A)$.

Sequents $\Gamma \Rightarrow \Delta$ have a finite multiset of formulas on each side. The inference rules for $\bigvee_{n>0}$ are thus:

$$\frac{\{\Gamma, A_n \Rightarrow \Delta \mid n > 0\}}{\Gamma, \bigvee_{n>0} A_n \Rightarrow \Delta} L\bigvee \qquad \frac{\Gamma \Rightarrow \Delta, \bigvee_{n>0} A_n, A_k}{\Gamma \Rightarrow \Delta, \bigvee_{n>0} A_n} R\bigvee_k.$$

Observe that $L\bigvee_{n>0}$ has countably many premisses, one for each $n > 0$. The rules for $\bigwedge_{n>0}$ are dual to the above ones.

Derivations built using these rules are thus (in general) infinite trees, with countable branching but where (as may be proved by induction on the definition of derivation) each branch has finite length. The *leaves* of the trees are those where the two sides have an atomic formula in common, and also instances of rules $L\perp$, $R\top$. To make this precise, we give a formal definition of the notion of *derivation* \mathcal{D} and the associated notions of its *height* $\text{ht}(\mathcal{D})$ and its *end-sequent*.

► **Definition 4** (Derivations, their height and their end-sequent).

1. Any sequent $\Gamma \Rightarrow \Delta$, where some atomic formula occurs in both Γ and Δ , is a derivation, of height 0 and with end-sequent $\Gamma \Rightarrow \Delta$.
2. Let $\beta \leq \omega$. If each \mathcal{D}_n , for $0 < n < \beta$, is a derivation of height α_n and with end-sequent $\Gamma_n \Rightarrow \Delta_n$, and

$$\frac{\dots \quad \Gamma_n \Rightarrow \Delta_n \quad \dots}{\Gamma \Rightarrow \Delta} R$$

is an instance of a rule with β premisses, then

$$\frac{\dots \quad \begin{array}{c} \vdots \mathcal{D}_n \\ \Gamma_n \Rightarrow \Delta_n \end{array} \quad \dots}{\Gamma \Rightarrow \Delta} R$$

is a derivation, of height the countable ordinal $\sup_n(\alpha_n) + 1$ and with end-sequent $\Gamma \Rightarrow \Delta$.⁴

If \mathbf{X} is a calculus, we use $\mathbf{X} \vdash \Gamma \Rightarrow \Delta$ to say that $\Gamma \Rightarrow \Delta$ is derivable in the calculus \mathbf{X} . By this definition each derivation has a countable ordinal *height* (the successor of the supremum of the heights of its immediate subderivations). Thus, if Γ and Δ have an atomic formula in common, then $\Gamma \Rightarrow \Delta$ has a derivation \mathcal{D} of height $\text{ht}(\mathcal{D}) = 0$. The sequent $\perp, \Gamma \Rightarrow \Delta$ (regarded as a zero-premiss rule) has a derivation of height 1. Observe that the definitions of depth of a formula and of height of a derivation differ from those in [6]: we use the successor of a supremum rather than the supremum of the successors (note that $\sup_{n>0}(n+1) = \omega \neq \omega + 1 = (\sup_{n>0}(n)) + 1$). It follows that, if \mathcal{D}' is a sub-derivation of \mathcal{D} , then $\text{ht}(\mathcal{D}') < \text{ht}(\mathcal{D})$. If a sequent has a derivation of height α we say it is α -*derivable* and write $\vdash^\alpha \Gamma \Rightarrow \Delta$.

⁴ Derivations can thus be represented as (infinite) trees, where the nodes are the sequents in the derivation, and a node that corresponds to a premiss of a rule is an immediate successor of the node that corresponds to the conclusion of such rule. Therefore, a node that corresponds to the conclusion of a rule with β premisses has β immediate successors.

► **Definition 5** (Sequent calculi for infinitary logics with equality).

1. $\mathbf{G3C}_\omega$ is defined by the rules in Tables 1 and 3;
2. $\mathbf{G3I}_\omega$ is defined as $\mathbf{G3C}_\omega$ with the exception of rules $L\supset$, $R\supset$, $R\forall$, and $R\wedge$ that are defined as in Table 2.

By $\mathbf{G3[CI]}_\omega$ we denote any one of the two calculi above. Observe that a multi-succedent intuitionistic calculus as the one we use is closer to a classical calculus than the usual calculus with the restriction that the succedent of sequents should consist of at most one formula (used, for example in [20]). As in the finitary case such a multi-succedent choice is particularly useful for proving Glivenko-style results [15].

As usual, we consider only derivations of *pure sequents* – i.e., sequents where no variable has both free and bound occurrences. We say that $\Gamma \Rightarrow \Delta$ is $\mathbf{G3[CI]}_\omega$ -*derivable* (with height α), and we write $\mathbf{G3[CI]}_\omega \vdash^{(\alpha)} \Gamma \Rightarrow \Delta$, if there is a $\mathbf{G3[CI]}_\omega$ -derivation (of height at most α) of $\Gamma \Rightarrow \Delta$ or of an alphabetic variant of $\Gamma \Rightarrow \Delta$. A rule is said to be (*height-preserving*) *admissible* in $\mathbf{G3[CI]}_\omega$, if, whenever its premisses are $\mathbf{G3[CI]}_\omega$ -derivable (with height at most α), also its conclusion is $\mathbf{G3[CI]}_\omega$ -derivable (with height at most α). A rule is said to be (*height-preserving*) *invertible* in $\mathbf{G3[CI]}_\omega$, if, whenever its conclusion is $\mathbf{G3[CI]}_\omega$ -derivable (with height at most α), also its premisses are $\mathbf{G3[CI]}_\omega$ -derivable (with height at most α). In each rule depicted in Tables 1, 2, and 3 the multisets Γ and Δ are called *contexts*, the formulas occurring in the conclusion are called *principal*, and the formulas occurring in the premiss(es) only are called *active*.

3 From geometric implications to geometric rules

By a *geometric implication* we mean the universal closure of an implicative formula whose antecedent and consequent are formulas constructed from atomic formulas and \perp, \top using only \wedge, \vee, \exists , and $\bigvee_{n>0}$. More precisely:

► **Definition 6** (Geometric implication).

- A formula is *Horn* iff it is built from atoms and \top using only \wedge ;
- A formula is *geometric* iff it is built from atoms and \top, \perp using only \wedge, \vee, \exists , and $\bigvee_{n>0}$;
- A sentence is a *geometric implication* iff it is of the form $\forall \vec{x}(A \supset B)$ where A and B are *geometric formulas*.

By a *coherent implication* we mean a geometric implication without occurrences of $\bigvee_{n>0}$.

As is well known, for geometric implications we have a normal form theorem.

► **Theorem 7** (Geometric normal form (GNF)). *Any geometric implication is equivalent to a possibly infinite conjunction of sentences of the form*

$$\forall \vec{x}(A \supset B)$$

where A is *Horn* and B is a possibly infinite disjunction of existentially quantified *Horn* formulas.

This normal form theorem is important because, as shown in [14] for coherent implications and in [16] for geometric ones, we can extract from a sentence G in GNF a *geometric rule* L_G (where the name L_G indicates that it is a *left rule*) that can be added to a sequent calculus without altering its structural properties. To be more precise, let us consider the following sentence G in GNF:

$$\forall \vec{x}(P_1(\vec{x}) \wedge \cdots \wedge P_k(\vec{x}) \supset \bigvee_{n>0} \exists \vec{y}(Q_{n_1}(\vec{x}, \vec{y}) \wedge \cdots \wedge Q_{n_m}(\vec{x}, \vec{y}))) \quad (G)$$

7:6 Constructive Cut Elimination in Geometric Logic

■ **Table 1** The calculus $\mathbf{G3C}_\omega$ (z fresh in $L\exists$ and $R\forall$).

Initial sequents: $P, \Gamma \Rightarrow \Delta, P$		
$\frac{}{\Gamma \Rightarrow \Delta, \top} R\top$	$\frac{\Gamma \Rightarrow \Delta, A \quad \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \wedge B} R\wedge$	$\frac{\Gamma \Rightarrow \Delta, A, B}{\Gamma \Rightarrow \Delta, A \vee B} R\vee$
$\frac{}{\perp, \Gamma \Rightarrow \Delta} L\perp$	$\frac{A, B, \Gamma \Rightarrow \Delta}{A \wedge B, \Gamma \Rightarrow \Delta} L\wedge$	$\frac{A, \Gamma \Rightarrow \Delta \quad B, \Gamma \Rightarrow \Delta}{A \vee B, \Gamma \Rightarrow \Delta} L\vee$
$\frac{A, \Gamma \Rightarrow \Delta, B}{\Gamma \Rightarrow \Delta, A \supset B} R\supset$	$\frac{\Gamma \Rightarrow \Delta, A(y/x), \exists x A}{\Gamma \Rightarrow \Delta, \exists x A} R\exists$	$\frac{\Gamma \Rightarrow \Delta, A(z/x)}{\Gamma \Rightarrow \Delta, \forall x A} R\forall$
$\frac{\Gamma \Rightarrow \Delta, A \quad B, \Gamma \Rightarrow \Delta}{A \supset B, \Gamma \Rightarrow \Delta} L\supset$	$\frac{A(z/x), \Gamma \Rightarrow \Delta}{\exists x A, \Gamma \Rightarrow \Delta} L\exists$	$\frac{A(y/x), \forall x A, \Gamma \Rightarrow \Delta}{\forall x A, \Gamma \Rightarrow \Delta} L\forall$
$\frac{\{\Gamma \Rightarrow \Delta, A_i \mid i > 0\}}{\Gamma \Rightarrow \Delta, \bigwedge_{n>0} A_n} R\bigwedge$	$\frac{\Gamma \Rightarrow \Delta, \bigvee_{n>0} A_n, A_k}{\Gamma \Rightarrow \Delta, \bigvee_{n>0} A_n} R\bigvee$	
$\frac{A_k, \bigwedge_{n>0} A_n, \Gamma \Rightarrow \Delta}{\bigwedge_{n>0} A_n, \Gamma \Rightarrow \Delta} L\bigwedge$	$\frac{\{A_i, \Gamma \Rightarrow \Delta \mid i > 0\}}{\bigvee_{n>0} A_n, \Gamma \Rightarrow \Delta} L\bigvee$	

■ **Table 2** Non-classical rules for $\mathbf{G3I}_\omega$ (z fresh in $R\forall$).

$\frac{A \supset B, \Gamma \Rightarrow \Delta, A \quad B, \Gamma \Rightarrow \Delta}{A \supset B, \Gamma \Rightarrow \Delta} L\supset$	$\frac{A, \Gamma \Rightarrow B}{\Gamma \Rightarrow \Delta, A \supset B} R\supset$	$\frac{\Gamma \Rightarrow A(z/x)}{\Gamma \Rightarrow \Delta, \forall x A} R\forall$	$\frac{\{\Gamma \Rightarrow A_i \mid i > 0\}}{\Gamma \Rightarrow \Delta, \bigwedge A_n} R\bigwedge$
--	---	---	---

■ **Table 3** Rules for equality in $\mathbf{G3[CI]}_\omega$.

$\frac{s = s, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} Ref$	$\frac{P(t/x), s = t, P(s/x), \Gamma \Rightarrow \Delta}{s = t, P(s/x), \Gamma \Rightarrow \Delta} Repl$
--	--

■ **Table 4** Geometric rule L_G expressing the geometric sentence G .

\dots	$Q_{n_1}(\vec{x}, \vec{y}_n), \dots, Q_{n_m}(\vec{x}, \vec{y}_n), P_1(\vec{x}), \dots, P_k(\vec{x}), \Gamma \Rightarrow \Delta$	\dots	L_G
	$P_1(\vec{x}), \dots, P_k(\vec{x}), \Gamma \Rightarrow \Delta$		

Such a sentence G determines the (finitary or infinitary) *geometric rule* given in Table 4 with one premiss for each of the countably many disjuncts in $\bigvee_{n>0} (Q_{n_1}(\vec{x}, \vec{y}) \wedge \dots \wedge Q_{n_m}(\vec{x}, \vec{y}))$. The variables in \vec{y}_n are chosen to be *fresh*, i.e. are not in the conclusion; and without loss of generality they are all distinct. The list \vec{y}_n of variables may vary as n varies, and maybe no finite list suffices for all the countably many cases. The variables \vec{x} (finite in number) may be instantiated with arbitrary terms. Henceforth we shall normally omit mention of the variables.

We need also a further condition for height-preserving admissibility of contraction to hold:

► **Definition 8 (Closure condition).** *Given a calculus with geometric rules, if it has a rule with an instance with repetition of some principal formula such as:*

$$\frac{\dots \quad Q_1, \dots, Q_n, P_1, \dots, P_{k-2}, P, P, \Gamma \Rightarrow \Delta \quad \dots}{P_1, \dots, P_{k-2}, P, P, \Gamma \Rightarrow \Delta} L_G^c$$

then also the contracted instance

$$\frac{\dots \quad Q_1, \dots, Q_m, P_1, \dots, P_{k-2}, P, \Gamma \Rightarrow \Delta \quad \dots}{P_1, \dots, P_{k-2}, P, \Gamma \Rightarrow \Delta} L_G^c$$

has to be included in the calculus.

As for the finitary case [14], also in the infinitary case the condition is unproblematic, since each atomic formula contains only a finite number of variables and therefore so are the instances; it follows that, for each geometric rule, the number of rules that have to be added is finite. Moreover, in many cases contracted instances need not be added since they are admissible in the calculus without them. To illustrate, we consider the coherent rule *Repl* for equality given in Table 3:

$$\frac{P(t/x), s = t, P(s/x), \Gamma \Rightarrow \Delta}{s = t, P(s/x), \Gamma \Rightarrow \Delta} Repl$$

This rule generates contracted instances only when its two principal formulas (as well as its active formula) are copies of the same reflexivity atom $t = t$. In this case, after having applied contraction, we can replace the instance of *Repl* with an instance of *Ref* (instead of *Repl*^c). That is, we can transform:

$$\frac{t = t, t = t, t = t, \Gamma \Rightarrow \Delta}{t = t, t = t, \Gamma \Rightarrow \Delta} Repl \quad \text{into} \quad \frac{\frac{t = t, t = t, t = t, \Gamma \Rightarrow \Delta}{t = t, t = t, \Gamma \Rightarrow \Delta} LC}{t = t, \Gamma \Rightarrow \Delta} Ref$$

But this does not hold in general. For example, if $<$ is an Euclidean relation, we must add both of the following rules:

$$\frac{s < r, t < s, t < r, \Gamma \Rightarrow \Delta}{t < s, t < r, \Gamma \Rightarrow \Delta} Euc \quad \text{and} \quad \frac{s < s, t < s, \Gamma \Rightarrow \Delta}{t < s, \Gamma \Rightarrow \Delta} Euc^c$$

otherwise the valid sequent $t < s \Rightarrow s < s$ would not be contraction-free derivable. In presence of *Ref*, no added rule is needed.

► **Theorem 9 ([16]).** *If we add to the calculus $\mathbf{G3[CI]}_\omega$ a finite or infinite family of geometric rules L_G , then we can prove all of the geometric sentences G from which they were determined.*

7:8 Constructive Cut Elimination in Geometric Logic

In the following, we shall denote with $\mathbf{G3[CI]_\omega.G}$ any extension of $\mathbf{G3[CI]_\omega}$ with a finite or infinite family of geometric rules L_G (together with all needed contracted instances thereof).

Before proceeding with the structural properties, we give some examples of geometric axioms and their corresponding rules.

► **Example 10** (Geometric axioms and rules).

1. The axiom of **torsion Abelian groups**, $\forall x. \bigvee_{n>1} (nx = 0)$, becomes the rule

$$\frac{\dots \quad nx = 0, \Gamma \Rightarrow \Delta \quad \dots}{\Gamma \Rightarrow \Delta} R_{Tor}$$

2. The axiom of **Archimedean ordered fields**, $\forall x. \bigvee_{n \geq 1} (x < n)$, becomes the rule

$$\frac{\dots \quad x < n, \Gamma \Rightarrow \Delta \quad \dots}{\Gamma \Rightarrow \Delta} R_{Arc}$$

3. The axiom of **connected graphs**,

$$\forall xy. x = y \vee xRy \vee \bigvee_{n>1} \exists z_0 \dots \exists z_n (x = z_0 \ \& \ y = z_n \ \& \ z_0Rz_1 \ \& \ \dots \ \& \ z_{n-1}Rz_n)$$

becomes the rule

$$\frac{x = y, \Gamma \Rightarrow \Delta \quad xRy, \Gamma \Rightarrow \Delta \quad \dots \quad x = z_0, y = z_n, z_0Rz_1, \dots, z_{n-1}Rz_n, \Gamma \Rightarrow \Delta \quad \dots}{\Gamma \Rightarrow \Delta} R_{Conn}$$

3.1 Structural rules

We present here the results concerning the admissibility of the structural rules, cut excluded, in the calculi $\mathbf{G3[CI]_\omega.G}$. All these results have been proved in Sect. 4 of [16] by simple transfinite induction on ordinals, either on the depth of a formula or on the height of a derivation.

► **Lemma 11** (Generalised initial sequents). *The sequent $A, \Gamma \Rightarrow \Delta$, A is $\mathbf{G3[CI]_\omega.G}$ -derivable, for A arbitrary formula.*

► **Lemma 12** (α -conversion). *If $\mathbf{G3[CI]_\omega.G} \vdash^\alpha \Gamma \Rightarrow \Delta$ then $\mathbf{G3[CI]_\omega.G} \vdash^\alpha \Gamma' \Rightarrow \Delta'$, for $\Gamma' \Rightarrow \Delta'$ a bound alphabetic variant of $\Gamma \Rightarrow \Delta$.*

► **Lemma 13** (Substitution). *If $\mathbf{G3[CI]_\omega.G} \vdash^\alpha \Gamma \Rightarrow \Delta$ and t is free for x in $\Gamma \Rightarrow \Delta$ then $\mathbf{G3[CI]_\omega.G} \vdash^\alpha \Gamma(t/x) \Rightarrow \Delta(t/x)$.*

► **Theorem 14** (Weakening). *The left and right rules of weakening:*

$$\frac{\Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta} LW \quad \frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta, A} RW$$

are height-preserving admissible (hp-admissible, for short) in $\mathbf{G3[CI]_\omega.G}$.

► **Lemma 15** (Invertibility).

1. Each rule of $\mathbf{G3C}_\omega.G$ is hp-invertible.
2. Each rule of $\mathbf{G3I}_\omega.G$ except $R\supset$, $R\forall$, and $R\wedge$ is hp-invertible.

► **Theorem 16** (Contraction). *The rules of left and right contraction:*

$$\frac{A, A, \Gamma \Rightarrow \Delta}{A, \Gamma \Rightarrow \Delta} LC \qquad \frac{\Gamma \Rightarrow \Delta, A, A}{\Gamma \Rightarrow \Delta, A} RC$$

are hp-admissible in $\mathbf{G3[CI]}_\omega.\mathbf{G}$.

4 Constructive cut elimination

We are now ready to prove that the following context-sharing rule of cut:

$$\frac{\Gamma \Rightarrow \Delta, C \quad C, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta} Cut$$

is eliminable in the calculus $\mathbf{G3[CI]}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ obtained by extending $\mathbf{G3[CI]}_\omega.\mathbf{G}$ with *Cut*. In order to give a proof of cut elimination that uses only constructively admissible proof-theoretical tools we must avoid the “natural” (or Hessenberg) commutative sum of ordinals: we cannot use cut-height as inductive parameter as done in the Gentzen- and Dragalin-style proofs. In order to avoid it, we make use of a proof strategy introduced in [13] for fuzzy logics that has been extensively used in the context of hypersequent calculi; see [4, 8, 10]. This proof strategy can be seen as a simplified and local version of the proof given by H.B. Curry in [5]. The proof is based on two main lemmas (Lemmas 20 and 21 below) that are proved by induction on the height of the derivation of the right and of the left premiss of cut, respectively. Moreover, (almost) all non-principal instances of cut are taken care of by a separate lemma (Lemmas 18 and 19) which shows that cut can be permuted upwards with respect to rule instances not having the cut formula among their principal formulas.

Observe that, differently from [4, 13, 10], we will not consider an arbitrary instance of *Cut* of maximal rank (i.e., such that its cut formula has maximal depth among the cut formulas occurring in the derivation), but we will always consider an uppermost instance of *Cut*, i.e. a cut the premisses of which are cut-free derivations. Otherwise, in Lemmas 20 and 21 as well as in Theorem 23, we would have to assume that ordinals are linearly/totally ordered; but in a constructive setting this assumption implies the law of excluded middle [1]. In Theorem 23 we will proceed, instead, by using two instances of Brouwer’s principle of Bar Induction: the first will be used to show that an uppermost instance of *Cut* is eliminable and the second to show that all instances of *Cut* are eliminable. Note that although it is a constructively admissible principle, Bar Induction increases the proof-theoretic strength of \mathbf{CZF} , cf. [19].

► **Definition 17** (Cut-substitutive rule). *A sequent rule Rule is cut-substitutive if each instance of cut with cut formula not principal in the last rule instance Rule of one of the premisses of cut can be permuted upwards w.r.t. Rule as in the following example:*

$$\frac{\frac{A, \Gamma \Rightarrow \Delta, B, C}{\Gamma \Rightarrow \Delta, A \supset B, C} R\supset \quad C, \Gamma \Rightarrow \Delta, A \supset B}{\Gamma \Rightarrow \Delta, A \supset B} Cut \quad \rightsquigarrow \quad \frac{\frac{A, \Gamma \Rightarrow \Delta, B, C}{\Gamma \Rightarrow \Delta, A \supset B, C} R\supset \quad \frac{C, \Gamma \Rightarrow \Delta, A \supset B}{A, C, \Gamma \Rightarrow \Delta, B} hp-inv}{\Gamma \Rightarrow \Delta, A \supset B} Cut$$

► **Lemma 18.** *Each rule of $\mathbf{G3C}_\omega.\mathbf{G}$ is cut-substitutive.*

Proof. By inspecting the rules in Tables 1 and 3 it is immediate to realise that each of them is cut-substitutive because they are all hp-invertible (using Lemma 13 for rules $L\exists$, $R\forall$, and for geometric rules with a variable condition). ◀

7:10 Constructive Cut Elimination in Geometric Logic

► **Lemma 19.** *Each rule of $\mathbf{G3I}_\omega.\mathbf{G}$ except $R\supset$, $R\forall$ and $R\wedge$ is cut-substitutive.*

Proof. Same as for $\mathbf{G3C}_\omega.\mathbf{G}$. ◀

► **Lemma 20** (Right reduction). *If we are in $\mathbf{G3[CI]}_\omega.\mathbf{G}$ and all of the following hold:*

1. $\mathcal{D}_1 \vdash \Gamma \Rightarrow \Delta, A$
2. $\mathcal{D}_2 \vdash A, \Gamma \Rightarrow \Delta$
3. A is principal in the last rule instance applied in \mathcal{D}_1
4. A is not of shape $\exists xB$ or $\bigvee_{n>0} B_n$.

Then there is a $\mathbf{G3[CI]}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ -derivation \mathcal{D} concluding $\Gamma \Rightarrow \Delta$ containing only cuts on proper subformulas of A .

Proof. By transfinite induction on $\text{ht}(\mathcal{D}_2)$.

If \mathcal{D}_2 is a one node tree, the lemma obviously holds.

Else, we have two cases depending on whether A is principal in the last rule instance $Rule$ applied in \mathcal{D}_2 or not.

In the latter case, if we are in $\mathbf{G3C}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$, the lemma holds thanks to Lemma 18: we permute the cut upwards in \mathcal{D}_2 and then we apply the inductive hypothesis and an instance of $Rule$. If we are in $\mathbf{G3I}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ and the last step of \mathcal{D}_2 is not by one of $R\supset$, $R\forall$, and $R\wedge$ then it holds by Lemma 19. In the remaining three cases, we have two subcases according to whether \mathcal{D}_1 ends with a step by an invertible rule or not. In the latter subcase, \mathcal{D}_1 ends with one of $R\supset$, $R\forall$, and $R\wedge$. We permute the cut upwards in the right premiss. To illustrate, we consider the case of $R\wedge$. We transform

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow B(y/x)}}{\Gamma \Rightarrow \Delta', \wedge A_n, \forall xB} R\forall \quad \frac{\frac{\vdots \mathcal{D}_{2i}}{\{\forall xB, \Gamma \Rightarrow A_i \mid i > 0\}}}{\forall xB, \Gamma \Rightarrow \Delta', \wedge A_n} R\wedge}{\Gamma \Rightarrow \Delta', \wedge A_n} Cut$$

into

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow B(y/x)}}{\Gamma \Rightarrow \forall xB} R\forall \quad \frac{\frac{\vdots \mathcal{D}_{2i}}{\{\forall xB, \Gamma \Rightarrow A_i \mid i > 0\}}}{\{\Gamma \Rightarrow A_i \mid i > 0\}} IH_i, i > 0}{\Gamma \Rightarrow \Delta', \wedge A_n} R\wedge$$

If, instead, \mathcal{D}_1 ends by an invertible rule then we apply invertibility, thus transforming the derivation into one having only cuts on proper subformulas of A . To illustrate, if \mathcal{D}_1 ends with a step by $R\wedge$, we transform

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta', \wedge A_n, B} \quad \frac{\vdots \mathcal{D}_{12}}{\Gamma \Rightarrow \Delta', \wedge A_n, C}}{\Gamma \Rightarrow \Delta', \wedge A_n, B \wedge C} R\wedge \quad \frac{\frac{\vdots \mathcal{D}_{2i}}{\{B \wedge C, \Gamma \Rightarrow A_i \mid i > 0\}}}{B \wedge C, \Gamma \Rightarrow \Delta', \wedge A_n} R\wedge}{\Gamma \Rightarrow \Delta', \wedge A_n} Cut$$

into

$$\frac{\frac{\frac{\vdots \mathcal{D}_{12}}{\Gamma \Rightarrow \Delta', \wedge A_n, C} \quad \frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta', \wedge A_n, B}}{C, \Gamma \Rightarrow \Delta', \wedge A_n, B} LW \quad \frac{\frac{\vdots \mathcal{D}_2}{B \wedge C, \Gamma \Rightarrow \Delta', \wedge A_n}}{B, C, \Gamma \Rightarrow \Delta', \wedge A_n} Lem. 15}}{C, \Gamma \Rightarrow \Delta', \wedge A_n} Cut}{\Gamma \Rightarrow \Delta', \wedge A_n} Cut$$

Next, we consider the case with A principal in the last rule instance applied in \mathcal{D}_2 . We have cases according to the shape of A .

If $A \equiv P$ for some atomic formula P , then the last rule instance in \mathcal{D}_2 is by a geometric rule (rules for equality included) L_G concluding $P_1, \dots, P, \dots, P_k, \Gamma'' \Rightarrow \Delta', P$ and \mathcal{D}_1 is the one node tree $P, \Gamma' \Rightarrow \Delta', P$. The conclusion of cut is the initial sequent $P, \Gamma' \Rightarrow \Delta', P$ which is cut-free derivable.

The cases with $A \equiv \perp$ or $A \equiv B \circ C$, for $\circ \in \{\top, \wedge, \vee, \supset\}$, are left to the reader.

If $A \equiv \forall x B$ we transform (if we are in $\mathbf{G3I}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$, Δ is not in the premiss of $R\forall$)

$$\frac{\frac{\begin{array}{c} \vdots \mathcal{D}_{11} \\ \Gamma \Rightarrow \Delta, B(y/x) \end{array}}{\Gamma \Rightarrow \Delta, \forall x B} R\forall \quad \frac{\begin{array}{c} \vdots \mathcal{D}_{21} \\ B(t/x), \forall x B, \Gamma \Rightarrow \Delta \end{array}}{\forall x B, \Gamma \Rightarrow \Delta} L\forall}{\Gamma \Rightarrow \Delta} Cut$$

into the following derivation having only cuts on proper subformulas of A (if we are in $\mathbf{G3I}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ then Δ is introduced in \mathcal{D}_{11} by height-preserving weakenings, which can be done since \mathcal{D}_{11} is in $\mathbf{G3I}_\omega.\mathbf{G}$):

$$\frac{\frac{\begin{array}{c} \vdots \mathcal{D}_{11} \\ \Gamma \Rightarrow \Delta, B(y/x) \end{array}}{\Gamma \Rightarrow \Delta, B(t/x)} Lem. 13 \quad \frac{\frac{\begin{array}{c} \vdots \mathcal{D}_1 \\ \Gamma \Rightarrow \Delta, \forall x B \end{array} \quad \frac{\begin{array}{c} \vdots \mathcal{D}_{21} \\ \forall x B, B(t/x), \Gamma \Rightarrow \Delta \end{array}}{B(t/x), \Gamma \Rightarrow \Delta} IH}{B(t/x), \Gamma \Rightarrow \Delta} Cut}{\Gamma \Rightarrow \Delta} Cut$$

If $A \equiv \bigwedge B_i$ we transform (Δ not in the premisses of $R\bigwedge$ if we are in $\mathbf{G3I}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$)

$$\frac{\frac{\begin{array}{c} \vdots \mathcal{D}_{1i} \\ \{\Gamma \Rightarrow \Delta, B_i \mid i > 0\} \end{array}}{\Gamma \Rightarrow \Delta, \bigwedge B_n} R\bigwedge \quad \frac{\begin{array}{c} \vdots \mathcal{D}_{21} \\ B_k, \bigwedge B_n, \Gamma \Rightarrow \Delta \end{array}}{\bigwedge B_n, \Gamma \Rightarrow \Delta} L\bigwedge}{\Gamma \Rightarrow \Delta} Cut$$

into the following derivation having only cuts on proper subformulas of A (if we are in $\mathbf{G3I}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ then Δ is introduced in \mathcal{D}_{1k} by height-preserving weakenings):

$$\frac{\frac{\begin{array}{c} \vdots \mathcal{D}_{1k} \\ \Gamma \Rightarrow \Delta, B_k \end{array} \quad \frac{\begin{array}{c} \vdots \mathcal{D}_1 \\ \Gamma \Rightarrow \Delta, \bigwedge B_n \end{array} \quad \frac{\begin{array}{c} \vdots \mathcal{D}_{21} \\ \bigwedge B_n, B_k, \Gamma \Rightarrow \Delta \end{array}}{B_k, \Gamma \Rightarrow \Delta} IH}{B_k, \Gamma \Rightarrow \Delta} Cut}{\Gamma \Rightarrow \Delta} Cut$$

► **Lemma 21** (Left reduction). *If we are in $\mathbf{G3[CI]}_\omega.\mathbf{G}$ and all of the following hold:*

1. $\mathcal{D}_1 \vdash \Gamma \Rightarrow \Delta, A$
2. $\mathcal{D}_2 \vdash A, \Gamma \Rightarrow \Delta$

Then there is a $\mathbf{G3[CI]}_\omega.\mathbf{G} + \{\mathbf{Cut}\}$ -derivation \mathcal{D} concluding $\Gamma \Rightarrow \Delta$ containing only cuts on proper subformulas of A .

Proof. By transfinite induction on $\text{ht}(\mathcal{D}_1)$.

If \mathcal{D}_1 is a one node tree, the lemma obviously holds.

Else, we have two cases depending on whether A is principal in the last rule instance applied in \mathcal{D}_1 or not. In the latter case, the lemma holds thanks to Lemma 18 or 19 (if the last step of \mathcal{D}_1 is by an intuitionistic non-invertible rule we proceed as in the analogous case of Lemma 20).

7:12 Constructive Cut Elimination in Geometric Logic

When A is principal in the last rule inference in \mathcal{D}_1 , we have cases according to the shape of A . If A is an atomic formula, or \top , or \perp , or $B \circ C$ ($\circ \in \{\wedge, \vee, \supset\}$), or $\forall xB$, or $\bigwedge B_n$, the lemma holds thanks to Lemma 20.

If $A \equiv \exists xB$, we transform:

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta, \exists xB, B(t/x)}}{\Gamma \Rightarrow \Delta, \exists xB} R\exists}{\Gamma \Rightarrow \Delta} \quad \frac{\vdots \mathcal{D}_2}{\exists xB, \Gamma \Rightarrow \Delta} \quad Cut}{\Gamma \Rightarrow \Delta}$$

into the following derivation having only cuts on proper subformulas of A (Lemma 15 can be applied since \mathcal{D}_2 is in $\mathbf{G3[CI]}_\omega.\mathbf{G}$):

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta, B(t/x), \exists xB}}{\Gamma \Rightarrow \Delta, B(t/x)} \quad \frac{\vdots \mathcal{D}_2}{\exists xB, \Gamma \Rightarrow \Delta} IH}{\Gamma \Rightarrow \Delta} \quad \frac{\frac{\vdots \mathcal{D}_2}{\exists xB, \Gamma \Rightarrow \Delta} \quad Lem. 15}{B(t/x), \Gamma \Rightarrow \Delta} \quad Cut}{\Gamma \Rightarrow \Delta}$$

If $A \equiv \bigvee_{n>0} B_n$, we transform:

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta, \bigvee_{n>0} B_n, B_k}}{\Gamma \Rightarrow \Delta, \bigvee_{n>0} B_n} R\bigvee}{\Gamma \Rightarrow \Delta} \quad \frac{\vdots \mathcal{D}_2}{\bigvee_{n>0} B_n, \Gamma \Rightarrow \Delta} \quad Cut}{\Gamma \Rightarrow \Delta}$$

into the following derivation:

$$\frac{\frac{\frac{\vdots \mathcal{D}_{11}}{\Gamma \Rightarrow \Delta, B_k, \bigvee_{n>0} B_n}}{\Gamma \Rightarrow \Delta, B_k} \quad \frac{\vdots \mathcal{D}_2}{\bigvee_{n>0} B_n, \Gamma \Rightarrow \Delta} IH}{\Gamma \Rightarrow \Delta} \quad \frac{\frac{\vdots \mathcal{D}_2}{\bigvee_{n>0} B_n, \Gamma \Rightarrow \Delta} \quad Lem. 15}{B_k, \Gamma \Rightarrow \Delta} \quad Cut}{\Gamma \Rightarrow \Delta} \quad \blacktriangleleft$$

In order to prove Cut elimination in a constructive way we use Bar Induction as done in [21, p. 18] for ω -arithmetic. This strategy avoids the assumption of total ordering of ordinal numbers. Before proving the theorem we introduce Brouwer's principle of (decidable) Bar Induction.

► **Definition 22** (Bar Induction). *Let B and I be unary predicates (the so-called “base predicate” and “inductive predicate”, respectively) of finite lists of natural numbers (to be denoted by u, v, \dots). If:*

1. B is decidable;
2. Every infinite sequence of natural numbers has a finite initial segment satisfying B ;
3. $B(u)$ implies $I(u)$ for every finite list u ;
4. If $I(u * n)$ holds for all $n \in \mathbb{N}$ then $I(u)$ holds;

Then I holds for the empty list of natural numbers.

► **Theorem 23** (Cut elimination). *Cut is admissible in $\mathbf{G3[CI]}_\omega$.*

Proof. Throughout this proof, we use finite lists of natural numbers to index (partial) branches of trees, i.e. directed paths from the root to a node, possibly a leaf. Consider a tree such that each node has immediate successors either indexed by ω or else by some $k < \omega$, and such that each branch has finite length, then:

- The empty list $\{\}$ indexes the root of the tree.
- Suppose that u indexes a partial branch \mathcal{R} of the tree and that the last node a has immediate successor nodes indexed by $k < \omega$, and let a natural number n be given. Let $m = n \bmod k$: that is, m is the remainder of n after division by k . Then $u * n$ indexes \mathcal{R} extended with the m^{th} immediate successor node of a . For example, in the case of a 2-premiss rule, odd numbers index the left premiss, even numbers the right premiss.

Notice that the above gives a partial surjective map, with decidable domain, from sequences of natural numbers to branches in the given tree. Moreover, this ensures that every infinite sequence has an initial segment that indexes a branch of the tree.⁵

Let \mathcal{D} be a derivation in the calculus $\mathbf{G3[CI]}_{\omega}.\mathbf{G} + \{\mathbf{Cut}\}$. The proof consists of two parts, each building on an appropriate Bar Induction.

- **Part 1.** We use Bar Induction to show that an uppermost instance of *Cut* with cut-formula C occurring in \mathcal{D} is admissible. We use the method defined above to index the branches of the formation tree of the formula C – where C is the root of the tree and atomic formulas or \top or \perp are its leaves. Let $B(u)$ hold if u indexes a branch whose last element is an atom or \perp or \top ; let $I(u)$ hold if u indexes a partial branch whose last element is a formula D such that an uppermost cut on D or on some proper subformula thereof in $\mathbf{G3[CI]}_{\omega}.\mathbf{G} + \{\mathbf{Cut}\}$ is eliminable.

The following hold:

1. $B(u)$ is decidable by simply comparing the list with the formation tree;
2. By definition of the indexing, the n^{th} element of the sequence identifies the n^{th} node in a branch of the formation tree of a formula. After a finite number of steps from the root we find an atom or \perp or \top since all branches of the tree are finite and this identifies an initial segment of the infinite sequence that satisfies B .
3. $B(u)$ implies $I(u)$ since cuts on atomic formulas, \top , or \perp are admissible;
4. $I(u * n)$ for all n implies $I(u)$: by Lemma 21 an uppermost cut on some formula E can be reduced to cuts on proper subformulas of E .

By Bar Induction we conclude that the uppermost cut with cut-formula C is eliminable from $\mathbf{G3[CI]}_{\omega}.\mathbf{G} + \{\mathbf{Cut}\}$.

- **Part 2.** We show that all cuts can be eliminated from \mathcal{D} . We consider a derivation \mathcal{D} in $\mathbf{G3[CI]}_{\omega}.\mathbf{G} + \{\mathbf{Cut}\}$ and, as above, we use lists of natural numbers to index branches of \mathcal{D} . Let $B(u)$ hold if u indexes a branch ending in a leaf of \mathcal{D} ; let $I(u)$ hold if u indexes a partial branch whose last element has a cut-free derivation (i.e., it is $\mathbf{G3[CI]}_{\omega}.\mathbf{G}$ -derivable). All conditions of Bar Induction are satisfied by this choice of B and I :

1. $B(u)$ is decidable;
2. Given any infinite sequence of numbers, we have $B(u)$ for every finite initial segment u that represents a full branch \mathcal{R} of the tree, i.e., a root-to-leaf path; and by construction of the representation there are such u .
3. $B(u)$ implies $I(u)$ since the leaves of \mathcal{D} trivially have a cut-free derivation;
4. $I(u * n)$ for all n implies $I(u)$: having shown in part 1 that uppermost instances of *Cut* are admissible, if all the premisses of a rule instance in \mathcal{D} have a cut-free derivation, then also its conclusion has a cut-free derivation.

By Bar Induction we conclude that the conclusion of \mathcal{D} has a cut-free derivation. ◀

⁵ Since the number of nodes of the tree is at most countable, one may also define an encoding such that the correspondence is unique. This however would require more effort and we would lose the property that every infinite sequence has an initial segment that indexes a branch of the tree.

► **Corollary 24.** *The rule of context-free cut:*

$$\frac{\Gamma \Rightarrow \Delta, A \quad A, \Pi \Rightarrow \Sigma}{\Pi, \Gamma \Rightarrow \Delta, \Sigma} \text{Cut}_{cf}$$

is admissible in $\mathbf{G3[CI]}_{\omega}.\mathbf{G}$.

Proof. An immediate consequence of Theorem 23 since rules Cut and Cut_{cf} are equivalent when weakening and contraction are admissible. ◀

5 A proof of the infinitary Barr theorem

Barr's theorem is a fundamental result in geometric logic: it guarantees that for geometric theories classical derivability of geometric implications entails their intuitionistic derivability. As recalled in the Introduction (Theorem 2), the result has its origin, through appropriate completeness results, in the theory of sheaf models. The most general form of Barr's theorem [3, 26, 20] is higher-order and includes the axiom of choice, and stated as eliminating not just classical reasoning but also the axiom of choice⁶.

If one is interested solely in derivability in geometric logic (finitary or infinitary, but without the axiom of choice), Barr's theorem can be regarded as identifying a Glivenko class, i.e., a class of sequents for which classical derivability entails intuitionistic derivability and a proof entirely internal to proof theory, without any detour through completeness with respect to topos-theoretic models, can be obtained.

Consider now a classical theory axiomatised by coherent or geometric implications. Extending the conversion into rules of [14] to cover the case of infinitary disjunctions and using the results detailed above, we transform the classical geometric theory \mathbf{G} into a contraction- and cut-free sequent calculus $\mathbf{G3C}_{\omega}.\mathbf{G}$. We shall denote by $\mathbf{G3I}_{\omega}.\mathbf{G}$ the corresponding intuitionistic extension of $\mathbf{G3I}_{\omega}.$ The following holds:

► **Theorem 25 (Barr's theorem).** *If a coherent or geometric implication is derivable in $\mathbf{G3C}_{\omega}.\mathbf{G}$, it is derivable in $\mathbf{G3I}_{\omega}.\mathbf{G}$.*

Proof. Any derivation in $\mathbf{G3C}_{\omega}.\mathbf{G}$ uses only rules that follow the (infinitary) geometric rule scheme and logical rules. Observe that geometric implications contain no \supset , nor \forall , nor \wedge in the scope of \vee , which means that no instance of the rules that violates the intuitionistic restrictions is used, so the derivation directly gives (through the addition, where needed, of the missing implications in steps of $L\supset$) a derivation in $\mathbf{G3I}_{\omega}.\mathbf{G}$ of the same conclusion. ◀

A proof of Barr's theorem for *finitary* geometric theories was given in [14] through a cut-free presentation of finitary geometric theories and the choice of an appropriate sequent calculus that, in effect, trivialises the result. By the results above, the same trivialization works for infinitary logic: a classical proof *already* is an intuitionistic proof.

6 Conclusion

This paper has shown how it is possible to constructivise the cut elimination procedure given in [16] for infinitary geometric theories and how, as a consequence, it is possible to obtain a constructive proof of Barr's theorem. The proof used here avoids the use of the natural sum

⁶ Cf. Footnote 1.

of ordinals which made non-constructive most cut-elimination procedures for infinitary logics, but it does not avoid the use of transfinite induction on ordinals since all proofs of the results in Section 3.1, as well as the proofs of Lemmas 20 and 21, are by induction on ordinals. We observe, however, that the alternative route of resorting to constructive ordinals has been pursued in [20] to obtain a proof of cut elimination for infinitary logic and of Barr’s theorem.

In the future, we plan to get rid of ordinals altogether by introducing a new well-founded inductive parameter that can supplant ordinals. Another open line of research is to extend the purely logical proof of Barr’s theorem given here and in [16] to other infinitary Glivenko sequent classes, as it has been done in [15] for the finitary ones.

References

- 1 Peter Aczel and Michael Rathjen. Notes on constructive set theory. Technical report, Institut Mittag-Leffler (The Royal Swedish Academy of Sciences), 2001. URL: <https://ncatlab.org/nlab/files/AczelRathjenCST.pdf>.
- 2 Ryota Akiyoshi. An ordinal-free proof of the complete cut-elimination theorem for $\Pi_1^1\text{-CA} + \text{BI}$ with the ω -rule. *IfCoLog J. of Logics and their Applications*, 4:867–884, January 2017.
- 3 Michael Barr. Toposes without points. *J. Pure Appl. Algebra*, 5(3):265–280, 1974. doi:10.1016/0022-4049(74)90037-1.
- 4 Agata Ciabattoni, George Metcalfe, and Franco Montagna. Algebraic and proof-theoretic characterizations of truth stressers for mtl and its extensions. *Fuzzy Sets and Systems*, 161(3):369–389, 2010. Fuzzy Logics and Related Structures. doi:10.1016/j.fss.2009.09.001.
- 5 Haskell B. Curry. *Foundations of Mathematical Logic*. Dover Books on Mathematics Series. Dover Publications, 1977.
- 6 Solomon Feferman. Lectures on Proof Theory. In *Proceedings of the Summer School in Logic (Leeds, 1967)*, pages 1–107. Springer, Berlin, 1968.
- 7 Peter Freyd. Aspects of topoi. *Bull. Austral. Math. Soc.*, 7(1):1–76, 1972. doi:10.1017/S0004972700044828.
- 8 Andrzej Indrzejczak. Eliminability of cut in hypersequent calculi for some modal logics of linear frames. *Information Processing Letters*, 115(2):75–81, 2015. doi:10.1016/j.ipl.2014.07.002.
- 9 Peter T. Johnstone. *Sketches of an Elephant: A Topos Theory Compendium. Vol. 1 & 2*. Oxford University Press, New York, 2002.
- 10 Hidenori Kurokawa. Hypersequent calculi for modal logics extending S4 . In Yukiko Nakano, Ken Satoh, and Daisuke Bekki, editors, *New Frontiers in Artificial Intelligence*, pages 51–68, Cham, 2014. Springer International Publishing. doi:10.1007/978-3-319-10061-6_4.
- 11 E. G. K. Lopez-Escobar. An interpolation theorem for denumerably long formulas. *Fund. Math.*, 57:253–272, 1965. doi:10.4064/fm-57-3-253-272.
- 12 Saunders Mac Lane and Ieke Moerdijk. *Sheaves in Geometry and Logic. A First Introduction to Topos Theory*. Springer-Verlag, 1994. doi:10.1007/978-1-4612-0927-0.
- 13 George Metcalfe, Nicola Olivetti, and Dov Gabbay. *Proof Theory for Fuzzy Logics*. Springer, 2008. doi:10.1007/978-1-4020-9409-5.
- 14 Sara Negri. Contraction-free sequent calculi for geometric theories with an application to Barr’s theorem. *Arch. Math. Logic*, 42(4):389–401, 2003. doi:10.1007/s001530100124.
- 15 Sara Negri. Glivenko sequent classes in the light of structural proof theory. *Arch. Math. Logic*, 55(3-4):461–473, 2016. doi:10.1007/s00153-016-0474-y.
- 16 Sara Negri. Geometric rules in infinitary logic. In Ofer Arieli and Anna Zamansky, editors, *Arnon Avron on Semantics and Proof Theory of Non-Classical Logics*, pages 265–293. Springer, 2021. doi:10.1007/978-3-030-71258-7_12.
- 17 Sara Negri and Jan von Plato. *Proof Analysis. A Contribution to Hilbert’s Last Problem*. Cambridge University Press, Cambridge, 2011.
- 18 V.P. Orevkov. Glivenko’s sequent classes. In V.P. Orevkov, editor, *Logical and logico-mathematical calculi. Part 1*, pages 131–154. Inst. Steklov, Leningrad, 1968.

7:16 Constructive Cut Elimination in Geometric Logic

- 19 Michael Rathjen. A note on Bar Induction in constructive set theory. *Mathematical Logic Quarterly*, 52(3):253–258, 2006. doi:10.1002/malq.200510030.
- 20 Michael Rathjen. Remarks on Barr’s theorem. Proofs in geometric theories. In Dieter Probst and Peter Schuster, editors, *Concepts of Proof in Mathematics, Philosophy, and Computer Science*, pages 347–374. de Gruyter, 2016. doi:10.1515/9781501502620-019.
- 21 Annika Siders. From Stenius’ consistency proof to Schütte’s cut elimination for ω -arithmetic. *Rev. Symb. Log.*, 9(1):1–22, 2016. doi:10.1017/S1755020315000337.
- 22 Göran Sundholm. Proof theory, a survey of the omega-rule. *Videnskapsselskapets skrifter, 1. Mat.-naturv. klasse*, 4, April 1983.
- 23 Gaisi Takeuti. *Proof Theory*. North-Holland, 1987 (second edition).
- 24 Anne S. Troelstra and Helmut Schwichtenberg. *Basic Proof Theory*. Cambridge University Press, Cambridge, 2nd edition, 2000. doi:10.1017/CB09781139168717.
- 25 Gavin C. Wraith. Generic Galois theory of local rings. In Michael P. Fourman et al., editor, *Applications of Sheaves*, pages 739–767. Springer-Verlag, 1979. doi:10.1007/BFb0061844.
- 26 Gavin C. Wraith. Intuitionistic algebra: some recent developments in topos theory. In *Proceedings of the International Congress of Mathematicians (Helsinki, 1978)*, pages 331–337. Acad. Sci. Fennica, 1980.

A Succinct Formalization of the Completeness of First-Order Logic

Asta Halkjær From   

Technical University of Denmark, Kongens Lyngby, Denmark

Abstract

I succinctly formalize the soundness and completeness of a small Hilbert system for first-order logic in the proof assistant Isabelle/HOL. The proof combines and details ideas from de Bruijn, Henkin, Herbrand, Hilbert, Hintikka, Lindenbaum, Smullyan and others in a novel way, and I use a declarative style, custom notation and proof automation to obtain a readable formalization. The formalized definitions of Hintikka sets and Herbrand structures allow open and closed formulas to be treated uniformly, making free variables a non-concern. This paper collects important techniques in mathematical logic in a way suited for both study and further work.

2012 ACM Subject Classification Theory of computation → Logic and verification

Keywords and phrases First-Order Logic, Completeness, Isabelle/HOL

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.8

Supplementary Material

Software (Formalization (stable)): https://isa-afp.org/entries/FOL_Axiomatic.html

Software (Formalization (latest)): https://devel.isa-afp.org/entries/FOL_Axiomatic.html

Acknowledgements Thanks to Frederik Krogsdal Jacobsen, Alexander Birch Jensen and Jørgen Villadsen for their useful comments. I am especially grateful to the anonymous reviewers for their insightful remarks which have improved both the formalization and the paper.

1 Introduction

The completeness of first-order logic has been known since Gödel’s work in 1929 [19]. Proof assistants too have a long history [18], with de Bruijn initiating the Automath project in 1967 and the first system of LCF, an Isabelle/HOL predecessor, being developed in 1972. Despite of this, I am unaware of a formalization of completeness in a proof assistant with a focus on explaining the core techniques. The ideas involved in such a proof deserve to be documented and detailed in a formalization where the proof assistant guarantees precision and correctness. This effort benefits students trying to understand mathematical logic and researchers looking for a base for their own work. I start from a Hilbert system, partly because I am unaware of a formalization which does the same, and partly because its simplicity allows me to focus on the ideas of the completeness proof itself. While other deduction systems may be more popular for first-order logic, Hilbert systems are still prominent in areas like modal logic.

This paper builds especially on work by Smullyan [40] and Henkin [21]. The Hilbert system of choice is Smullyan’s System Q1 [40, p. 81] and the completeness proof resembles the “more direct construction” near the end of his book [40, p. 96] (a construction that was pointed out to him by Henkin himself). This paper formalizes ideas by de Bruijn, Henkin, Herbrand [23], Hilbert, Hintikka, Lindenbaum and Smullyan in an attempt to give a “strikingly direct” [40, p. 96] completeness proof formalized in a modern proof assistant.

Smullyan includes a generalization rule for the universal quantifier that works under an assumption (i.e. to the right of an implication) rather than on a standalone formula. This extra generality makes it very suited for my purposes, where I always work under a number



© Asta Halkjær From;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 8; pp. 8:1–8:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

of assumptions. Smullyan does not consider function symbols as part of his syntax, but his System Q1 is easily extended to cover these: simply allow for any term in the instantiation axiom. Barwise [1] makes the same modification.

I use the proof assistant Isabelle/HOL [34]. Isabelle is a generic proof assistant and Isabelle/HOL is the version based on higher-order logic. This paper includes a number of Isabelle listings, all taken from the formalization after it has been exported to L^AT_EX. These listings appear either in bulleted lists or prefixed by an Isabelle command in **bold** and should therefore be clearly distinguishable from the surrounding text. Any such listing has been checked and verified by the proof assistant. I sometimes use these listings to explain proofs. In these cases, I do not include the commands that justify each step of reasoning, but only the intermediate statements themselves. For clarity, I have omitted many types from the main text. Some of these can be found in Table 1 on page 6. The full formalization (under 700 lines) is available in the Archive of Formal Proofs [17], which referees Isabelle formalizations and, when accepted, keeps them up to date with the latest version of the proof assistant.

Contributions

The main contribution of this paper is a succinct formalization of the definitions and proofs that make up the synthetic style, a widely applicable method of proving completeness.

As a smaller contribution, this is, to my knowledge, the first formalization of completeness for classical first-order logic that starts from a Hilbert system. However, several formalizations that start from other proof systems are available (cf. Section 2) and the relations between various proof systems have also been formalized, see for instance recent work by Laurent [27] in Coq on translating between Hilbert systems and natural deduction for first-order logic.

On the more technical side, I formalize a Herbrand universe which, like in Herbelin and Ilik’s [22] unformalized proof, consists of all terms, not just those without variables. Combined with a Hintikka set in the style of Forster et al. [11] in Coq, based on the absence of formulas rather than the presence of their negations, but which, unlike theirs, contains open formulas as well as closed, I naturally formalize completeness for all valid formulas.

Isabelle/HOL Overview

This section seeks to give a quick overview of the Isabelle/HOL features used later. Nipkow and Klein [33, Part 1] give a more complete introduction.

The higher-order logic of Isabelle/HOL can be reasonably understood as functional programming + logic [33]. The **datatype** command defines a new type from a series of constructors, where each can be given custom syntax. The natural numbers are built from the nullary constructor *0* and unary *Suc*. The constructors *True* and *False* belong to the built-in type *bool*. The usual connectives and quantifiers from first-order logic (\longrightarrow , \forall , etc.) are available for *bool*, as well as *if-then-else* expressions. We can write total functions over datatypes using **primrec** for primitive recursive functions and **fun** for more advanced definitions. The type constructor $A \Rightarrow B$ denotes a function from *A* to *B*. Instead of concrete types, we can also use type variables *'a*, *'b*, etc. These stand in the place of other types. The term *UNIV* stands for the set of all values of a given type.

Another important built-in type is *'a list*, the type of lists whose elements are of type *'a*. These are built from $[\]$, the empty list, and $\#$, an infix constructor that adjoins an element to an existing list. The notation $[a, b, c]$ is shorthand for these primitive operations. The function *set* turns a list into a set of its elements. The higher-order function *map* applies a given function to every element of a list.

Function application resembles functional programming languages in that $f(x, y)$ is written as $f x y$. The function $f(x := y)$ maps x to y and every other element x' to $f x'$. Anonymous functions can be built using λ -expressions, e.g. $\lambda n. n + n$ for $f(n) = n + n$.

In proofs, the meta-logical implication \implies separates assumptions from conclusions. These can also be distinguished using the **assumes** and **shows** keywords, using **and** as a separator when there are multiple assumptions or conclusions. The keyword **have** states an intermediate fact in a proof and the keywords **then**, **moreover** and **ultimately** bind these statements together in different ways. The keyword **let** introduces a local abbreviation and **obtain** eliminates an existential statement; **for** quantifies a statement universally.

The command **definition** introduces a new definition that is hidden behind a name and must be explicitly unfolded, while an **abbreviation** is unfolded by the parser. The **inductive** command also makes use of the meta-logical implication. This command allows us to specify the least predicate closed under some given rules. For instance a predicate that denotes whether a formula can be derived, specified by axioms and inference rules. A **locale** in Isabelle, as used here, names an association between terms and assumptions about them. We could, for instance, specify a group as a set and a binary operation coupled with the group axioms. To instantiate the locale we must give concrete terms and show that they satisfy the assumptions. When assuming a locale, we assume the conditions hold for the terms.

The axiom of choice is available as Hilbert's choice operator: the expression $SOME x. P x$ returns some element x that satisfies the predicate P , when such an element exists.

Overview of Paper

I discuss related work next (Section 2). In Section 3, I formalize the syntax of first-order logic in Isabelle/HOL, including the idea of de Bruijn indices. This idea is developed further in Section 4 on the semantics of terms and formulas. Section 5 formalizes the proof system and its soundness, including all the operations necessary to do so. This includes the instantiation of universal quantifiers, propositional tautologies and a range of lemmas. I prove the completeness in Section 6 where I introduce the notion of a maximal consistent set, the Lindenbaum construction and the model existence theorem for Hintikka sets. I discuss challenges and choices in Section 7 and conclude with thoughts on future work in Section 8.

2 Related Work

The completeness of first-order logic itself has a long history, starting with Gödel's proof [19] and Henkin's later simplification [21]. Smullyan [40], Barwise [1] and Fitting [10] all include completeness proofs in their texts. Smullyan's main completeness proof is an "analytic" proof for a tableau system. He devotes only two pages to the "synthetic" (also called Henkin-style) completeness proof of System Q1 [40, pp. 96–97] that I formalize a version of here. Barwise [1] considers System Q1 extended with axioms for equality and gives a quite different proof that relies on a reduction to propositional logic (and the completeness of propositional logic). Fitting [10] proves completeness for tableaux and resolution similarly to Smullyan and leaves the completeness of his Hilbert system as an exercise for the reader. This paper spells out the synthetic completeness proof for first-order logic, starting from a Hilbert system rather than from tableaux, resolution or similar.

The synthetic style has been successfully applied in several formalizations lately. From [13] used it to formalize the completeness of a Hilbert system for propositional logic in Isabelle/HOL. Berghofer [3] formalized natural deduction for first-order logic in Isabelle/HOL

following the work by Fitting [10]. My formalization of the syntax and semantics of first-order logic and the Lindenbaum construction is inspired by his work. My formalization of Hintikka sets and proof of the model existence theorem, however, differ from his and is inspired by Herbelin and Ilik [22] and Forster et al. [11]. In particular, I prove completeness for open and closed formulas together, where Berghofer’s completeness proof only covers closed formulas and is extended to cover open formulas afterwards. From, Schlichtkrull and Villadsen [14, 16] built on Berghofer’s work to formalize the completeness of both a sequent calculus and tableau system for first-order logic. They also described Berghofer’s formalization in detail. Bentzen [2] formalized the completeness of a Hilbert system for the modal logic S5 in Lean. Jørgensen et al. [26] gave a synthetic completeness proof for a tableau system for basic hybrid logic, which From [12, 15] formalized in Isabelle/HOL.

I am far from the first to formalize the completeness of first-order logic, but my formalization is the only one that proves completeness for a Hilbert system for classical first-order logic. Shankar [39] formalized a tautology checker for first-order logic in the Boyer-Moore theorem prover, but notably did not formalize first-order completeness. Harrison [20] also formalized first-order logic in higher-order logic (but HOL rather than Isabelle/HOL). He did not formalize a proof system but rather model-theoretic results like compactness and the Löwenheim-Skolem theorem. Margetson and Ridge [29] formalized the completeness of first-order logic without functions in Isabelle/HOL via a sequent calculus. Braselmann and Koepke [7] did the same in their Mizar formalization. Schlichtkrull [37, 38] formalized the completeness of first-order resolution in Isabelle/HOL. Michaelis and Nipkow [30, 31] did not formalize first-order logic, but did formalize a Hilbert system for propositional logic in Isabelle/HOL. They proved completeness via translation from a sequent calculus with an analytic completeness proof. Blanchette, Popescu and Traytel [5, 6] formalized analytic completeness of abstract sequent calculus and tableau systems for first-order logic in Isabelle/HOL. Blanchette [4] outlines formalizations of logical meta-theory in Isabelle/HOL.

The completeness proof presented here relies on Lindenbaum’s lemma: that any consistent set of formulas has a maximal consistent extension. The proof is non-constructive since, for the given semantics, Lindenbaum’s lemma is equivalent to Weak König’s Lemma [22, 24]. There are, however, a number of formalizations of completeness in other meta-theories (and necessarily using other semantics). Veldman [43] gave an intuitionistic completeness theorem for intuitionistic predicate logic in 1976. Persson [35] formalized the completeness of sequent calculus and natural deduction for intuitionistic first-order logic in the ALF proof assistant, but only defined a Hilbert system without further proof. His models are based on formal topology. Constable and Bickford [8] constructively proved completeness for intuitionistic first-order logic in the proof assistant Nuprl. Ilik [25] formalized multiple constructive proofs of first-order completeness in the proof assistant Coq using novel variants of Kripke models for full classical and intuitionistic first-order logic. Forster et al. [11] recently analyzed the computational content of completeness theorems for various semantics and for natural deduction and sequent calculus systems. They mechanized their results in constructive type theory using Coq.

3 Syntax

The following syntax will be our starting point.

A term t is either a variable x or a function symbol f applied to a number of other terms:

$$s, t ::= x \mid f(t_1, \dots, t_n)$$

A function symbol applied to zero terms is called a constant and is denoted by a .

A formula p is either falsity (denoted \perp), a predicate symbol P applied to a list of terms, an implication (\longrightarrow) between two formulas or a universally quantified formula:

$$p, q ::= \perp \mid P(t_1, \dots, t_n) \mid p \longrightarrow q \mid \forall x. p(x)$$

I apply a number of techniques to make this syntax suitable for formalization. First, I represent the variables with de Bruijn indices [9]. Instead of connecting quantifiers and variables by using the same variable symbol x , each variable is a natural number n that is understood to be connected to the n th quantifier, starting from the innermost. For example, the formula $\forall x. \forall y. P(x, y)$ is represented as $\forall \forall P(1, 0)$. This makes it simpler to define capture-avoiding instantiation, which we need for the proof system.

Second, to distinguish variables, function symbols and predicate symbols in the proof assistant, I prefix each kind by a distinct symbol: \dagger for function symbols, \ddagger for predicate symbols and $\#$ for variables. The formula $\forall P(f(0))$ is then written (for now) as $\forall \ddagger P(\dagger f(\#0))$.

Finally, lists of argument terms are represented as proper Isabelle/HOL lists, so the term $f(a)$ becomes $\dagger f [a]$.

The (parameterized) datatype $'f \text{ tm}$ of terms with function symbols of type $'f$ becomes:

```
datatype (params-tm: 'f) tm
  = Var nat (#)
  | Fun 'f ('f tm list) ( $\dagger$ )
```

The annotation *params-tm* generates a function of that name from terms to $'f$ sets: it collects all values of type $'f$ in a given term. I discuss its properties in Section 5.1.

The following abbreviates a constant, as I use these frequently:

```
abbreviation Const ( $\star$ ) where  $\star a \equiv \dagger a []$ 
```

The datatype $('f, 'p) \text{ fm}$ of formulas with functions symbols of type $'f$ and predicate symbols of type $'p$ becomes:

```
datatype (params-fm: 'f, 'p) fm
  = Falsity ( $\perp$ )
  | Pre 'p ('f tm list) ( $\ddagger$ )
  | Imp (( 'f, 'p) fm) (( 'f, 'p) fm) (infixr  $\longrightarrow$  55)
  | Uni (( 'f, 'p) fm) ( $\forall$ )
```

The custom notation for each syntactic case is enclosed in parentheses (**infixr** specifies right associativity and 55 specifies parsing priority). I use bold symbols to avoid conflicts with existing syntax. The notation *params-fm*, similarly to for terms, generates a function which produces a set of all function symbols in a given formula.

The Isabelle command **term** checks the type of an expression. Given the above definitions, we can try our syntax, here with strings for the types of function and predicate symbols:

```
term  $\forall (\perp \longrightarrow \ddagger''P'' [\dagger''f'' [\#0]])$ 
```

In regular notation this would be $\forall x. \perp \longrightarrow P(f(x))$.

The following abbreviation for negation will ease notation: $\neg p \equiv p \longrightarrow \perp$.

Similar notations can easily be introduced for conjunction, disjunction, the existential quantifier etc. since in classical logic, these can be defined using the given syntax.

It should be noted that since arities are implicit in the datatypes above, we unfortunately cannot represent finite signatures. The awarded benefit is that we do not need a predicate to distinguish between correct and incorrect syntax.

■ **Table 1** Type signatures for selected functions.

<i>semantics-tm</i>	$(\text{nat} \Rightarrow 'a) \Rightarrow ('f \Rightarrow 'a \text{ list} \Rightarrow 'a) \Rightarrow 'f \text{ tm} \Rightarrow 'a$
<i>semantics-fm</i>	$(\text{nat} \Rightarrow 'a) \Rightarrow ('f \Rightarrow 'a \text{ list} \Rightarrow 'a) \Rightarrow ('p \Rightarrow 'a \text{ list} \Rightarrow \text{bool}) \Rightarrow ('f, 'p) \text{ fm} \Rightarrow \text{bool}$
<i>shift</i>	$(\text{nat} \Rightarrow 'a) \Rightarrow \text{nat} \Rightarrow 'a \Rightarrow \text{nat} \Rightarrow 'a$
<i>boolean</i>	$('p \Rightarrow 'f \text{ tm list} \Rightarrow \text{bool}) \Rightarrow (('f, 'p) \text{ fm} \Rightarrow \text{bool}) \Rightarrow ('f, 'p) \text{ fm} \Rightarrow \text{bool}$
<i>Axiomatic</i>	$('f, 'p) \text{ fm} \Rightarrow \text{bool}$
<i>imply</i>	$('f, 'p) \text{ fm list} \Rightarrow ('f, 'p) \text{ fm} \Rightarrow ('f, 'p) \text{ fm}$
<i>consistent</i>	$('f, 'p) \text{ fm set} \Rightarrow \text{bool}$
<i>extend</i>	$('f, 'p) \text{ fm set} \Rightarrow (\text{nat} \Rightarrow ('f, 'p) \text{ fm}) \Rightarrow \text{nat} \Rightarrow ('f, 'p) \text{ fm set}$
<i>witness</i>	$'f \text{ set} \Rightarrow ('f, 'p) \text{ fm} \Rightarrow ('f, 'p) \text{ fm set}$
<i>hmodel</i>	$('f, 'p) \text{ fm set} \Rightarrow ('f, 'p) \text{ fm} \Rightarrow \text{bool}$

4 Semantics

The semantics of first-order logic has two parts: one for terms and one for formulas. I formalize both as functions.

4.1 Terms

A term evaluates to an element of the *domain*. It does so under an *environment* (a mapping from variables to the domain) and a *function denotation* (a mapping from function symbols to functions on the domain).

In Isabelle, I represent the domain as a type (variable) and the environment as a function E from the natural numbers (the de Bruijn indices) to that type. Similarly, the function denotation becomes the function F from function symbols to functions on the domain. This results in the following definition:

primrec *semantics-tm* ($([-, -])$) **where**
 $([E, F]) (\#n) = E n$
 $| ([E, F]) (\dagger f \text{ ts}) = F f (\text{map } ([E, F]) \text{ ts})$

The semantics of a variable is given by the environment and in the case of a function application $\dagger f \text{ ts}$, we must first evaluate all the argument terms ts (using *map*) and then apply the function denoted by f .

Here $([E, F])$ denotes the function from terms to the domain, given by the environment E and function denotation F . As seen in the clause above for functions, this notation lets me conveniently “bundle” a given E and F so they can be applied to any term without the need for anonymous functions. I use a similar notation $[[E, F, G]]$ for the semantics of formulas.

4.2 Formulas

I use a deep embedding where formulas evaluate to a truth value under an environment E , a function denotation F and a *predicate denotation*, dubbed G , that maps predicate symbols to predicates on the domain. I formalize it as follows:

primrec *semantics-fm* ($[[-, -, -]]$) **where**
 $[[-, -, -]] \perp = \text{False}$
 $| [[E, F, G]] (\dagger P \text{ ts}) = G P (\text{map } ([E, F]) \text{ ts})$
 $| [[E, F, G]] (p \longrightarrow q) = ([[E, F, G]] p \longrightarrow [[E, F, G]] q)$
 $| [[E, F, G]] (\forall p) = (\forall x. [[E(\theta:x), F, G]] p)$

The formula \perp is always *False* and the semantics of a predicate is similar to that of a function application. For implication each subformula is evaluated to a truth value and the connective is interpreted using the built-in implication. Similarly, I use the built-in universal quantifier to interpret the object-level quantifier. The notation $E\langle 0:x \rangle$ is explained next.

4.3 Shifting

The expression $E\langle n:x \rangle$ modifies the environment E such that variable n is assigned x , any smaller variable m is assigned $E\ m$ and any larger variable m is assigned $E\ (m - 1)$. This *shift* operation has the following definition:

definition *shift* ($- \langle \cdot : \cdot \rangle$) **where**
 $E\langle n:x \rangle\ m \equiv$ if $m < n$ then $E\ m$ else if $m = n$ then x else $E\ (m - 1)$

To understand the shifting operation on larger variables, consider the following:

$$\llbracket E, F, G \rrbracket (\forall \forall \ddagger P [\#0, \#1])$$

By the semantics, this reduces to:

$$\forall x. \llbracket E\langle 0:x \rangle, F, G \rrbracket (\forall \ddagger P [\#0, \#1])$$

where the outer quantifier comes from the meta-logic. This again reduces to:

$$\forall x. \forall y. \llbracket E\langle 0:x \rangle\langle 0:y \rangle, F, G \rrbracket (\ddagger P [\#0, \#1])$$

Thus, the terms are evaluated by $\llbracket E\langle 0:x \rangle\langle 0:y \rangle, F \rrbracket$. This is clearly correct for variable $\#0$ since $E\langle 0:x \rangle\langle 0:y \rangle\ 0 = y$ as desired. We also want that $\#1$ corresponds to the outer meta-logic quantifier, namely that $E\langle 0:x \rangle\langle 0:y \rangle\ 1 = x$. This is exactly what happens since $E\langle 0:x \rangle\langle 0:y \rangle\ 1 = E\langle 0:x \rangle\ (1 - 1) = x$. Thus, the semantics reduces to the expected:

$$\forall x. \forall y. G\ P\ [y, x]$$

Notice that any *free* variable in a formula (those with no corresponding quantifier) are not affected by this shifting when it is coupled with the removal of an outer quantifier: they are mapped to whatever E originally assigned them to. In this sense they behave like constants.

The following four lemmas will be used implicitly.

► **Lemma 1 (Shifting).** *The first three results characterize the function and the last one commutes a shift of variable 0 with another shift.*

- $n = m \implies E\langle n:x \rangle\ m = x$
- $m < n \implies E\langle n:x \rangle\ m = E\ m$
- $n < m \implies E\langle n:x \rangle\ m = E\ (m - 1)$
- $(E\langle n:y \rangle\langle 0:x \rangle) = (E\langle 0:x \rangle\langle n+1:y \rangle)$

Proof. Immediate from the definition. ◀

5 Proof System

To define the proof system I must first define a number of operations needed to state the side conditions and transformations of formulas.

5.1 Parameters

The proof rule for the universal quantifier will generalize a *fresh* constant to a quantified variable. To perform this freshness check, I use the functions *params-tm* and *params-fm* that Isabelle generates automatically from the datatype declarations above. These collect all function symbols in terms and formulas, respectively, and would also be easy to define recursively. Similarly to Smullyan [40], I abbreviate function symbol to *parameter*.

The following definition generalizes *params-fm* to a set of formulas:

abbreviation $\text{params } S \equiv \bigcup p \in S. \text{params-fm } p$

I need a few lemmas about parameters for later.

► **Lemma 2** (Finite parameters). *Terms and formulas contain only finitely many parameters:*

- *finite* (*params-tm* t)
- *finite* (*params-fm* p)

Proof. By simple inductions. ◀

► **Lemma 3** (Unused parameters). *The denotation of an unused parameter does not affect the semantics of either terms or formulas:*

- $f \notin \text{params-tm } t \implies \langle E, F(f := x) \rangle t = \langle E, F \rangle t$
- $f \notin \text{params-fm } p \implies \llbracket E, F(f := x), G \rrbracket p = \llbracket E, F, G \rrbracket p$

Proof. By simple inductions. ◀

5.2 Instantiation

I will need to instantiate a universally quantified formula with a concrete term: to go from $\forall p$ to “ p with t inserted for variable 0 and free variables in p adjusted.” I will denote this formula by $\langle t/0 \rangle p$. Note that when instantiating for n in $\forall p$, we need to then instantiate for $n+1$ in p , since we enter the scope of another quantifier (the formula $\forall x. \forall y. P(x, y)$ becomes $\forall \forall P(1, 0)$ with de Bruijn indices, so to instantiate for x we must actually replace variable 1).

There are two additional considerations. Consider first why we need to adjust the free variables in p . Say that we are instantiating $\forall P [\#0, \#3]$ with the term t . When evaluating $\forall P [\#0, \#3]$ under an environment E , the free variable 3 will be interpreted as $(E \langle 0:x \rangle) 3 = E 2$. We expect that the interpretation of free variables under the same environment does not change when we instantiate a quantifier. However, when evaluating the naively instantiated formula $P [t, \#3]$, the free variable 3 will be evaluated as $E 3$, which might be a different value than $E 2$. Therefore, we should decrement any free variables we encounter during the instantiation. The result here should then be $P [t, \#2]$.

Second, it is important that any free variable in t remains free in $\langle t/0 \rangle p$ (i.e. that the instantiation avoids capturing a free variable). With named variables we would ensure this by renaming any bound variables in p that would conflict. By using de Bruijn indices we are free from having to come up with fresh names for such an operation. Instead, we increment every variable in t by one whenever we pass under a quantifier. Thus $\langle \uparrow f [\#0]/0 \rangle (\forall (\ddagger P)) = \forall (\langle \uparrow f [\#1]/1 \rangle (\ddagger P))$.

I call this last operation *lifting* the term:

primrec *lift-tm* (\uparrow) **where**
 $\uparrow (\#n) = \#(n+1)$
 $\uparrow (\uparrow f ts) = \uparrow f (\text{map } \uparrow ts)$

While this terminology is common (cf. Nipkow [32], Berghofer [3]) it unfortunately conflicts with the terminology in explicit substitutions (cf. Lescanne [28]) where *lifting* and *shifting* have roughly opposite meanings compared to this paper.

With the above considerations in mind, we can now define instantiation on terms:

primrec *inst-tm* ($\langle\langle\cdot/\cdot\rangle\rangle$) **where**
 $\langle\langle s/m\rangle\rangle(\#n) = (\text{if } n < m \text{ then } \#n \text{ else if } n = m \text{ then } s \text{ else } \#(n-1))$
 $|\langle\langle s/m\rangle\rangle(\dagger f \ ts) = \dagger f \ (\text{map } \langle\langle s/m\rangle\rangle \ ts)$

The notation $\langle\langle s/m\rangle\rangle$ “bundles” an instantiation of term s for variable m , ready to be applied to a term. For formulas, the only interesting case is for the universal quantifier, where we lift the term we are instantiating with and increment the variable we are instantiating for:

primrec *inst-fm* ($\langle\langle\cdot/\cdot\rangle\rangle$) **where**
 $\langle\langle\cdot/\cdot\rangle\rangle\perp = \perp$
 $|\langle\langle s/m\rangle\rangle(\dagger P \ ts) = \dagger P \ (\text{map } \langle\langle s/m\rangle\rangle \ ts)$
 $|\langle\langle s/m\rangle\rangle(p \longrightarrow q) = \langle\langle s/m\rangle\rangle p \longrightarrow \langle\langle s/m\rangle\rangle q$
 $|\langle\langle s/m\rangle\rangle(\forall p) = \forall(\langle\langle \uparrow s/m+1\rangle\rangle p)$

Despite the complexity of instantiation when using de Bruijn indices, it can be captured in the three simple definitions above that involve little more than simple arithmetic.

A more standard name for $\langle\langle t/n\rangle\rangle p$ is substitution, but I prefer instantiation since it potentially does more than simple syntactic substitution of term t for variable n : namely lifts t and decrements variables in p .

The only results about instantiation that I need for the formalization are the following.

► **Lemma 4** (Lifting and shifting). *Lifting cancels out with shifting the environment at 0.*

■ $\langle\langle E\langle 0:x\rangle, F\rangle\rangle(\uparrow t) = \langle\langle E, F\rangle\rangle t$

Proof. By structural induction. ◀

► **Lemma 5** (Instantiation and shifting). *Instantiating with a term at m is the same as shifting the environment at m with the value denoted by that term.*

■ $\langle\langle E, F\rangle\rangle(\langle\langle s/m\rangle\rangle t) = \langle\langle E\langle m:\langle\langle E, F\rangle\rangle s, F\rangle\rangle t$

■ $\llbracket E, F, G \rrbracket(\langle\langle t/m\rangle\rangle p) = \llbracket E\langle m:\langle\langle E, F\rangle\rangle t, F, G \rrbracket p$

Proof. By structural induction, using Lemma 4. ◀

5.3 Size

To prove the model existence theorem, I will need to do induction on formulas. However, structural induction does not work, since in the case for $\forall p$, the induction hypothesis must be applied to the instance $\langle\langle t/0\rangle\rangle p$, for some term t , rather than simply to p . This calls for induction on the size of the formula. Unfortunately, the pre-defined *size* measure for our datatype takes the size of terms into account and is therefore not invariant under instantiation. The following definition suffices:

primrec *size-fm* **where**
 $\text{size-fm } \perp = 1$
 $|\text{size-fm } (\dagger \cdot) = 1$
 $|\text{size-fm } (p \longrightarrow q) = 1 + \text{size-fm } p + \text{size-fm } q$
 $|\text{size-fm } (\forall p) = 1 + \text{size-fm } p$

► **Lemma 6** (Size). *Instantiation preserves size.*

■ $\text{size-fm } (\langle\langle t/m\rangle\rangle p) = \text{size-fm } p$

Proof. By structural induction. ◀

5.4 Propositional Semantics

Instead of picking a suitable set of propositional axioms, Smullyan [40], Barwise [1] and others simply include all tautologies as one of their axioms. I follow their lead and need a suitable way to express which formulas are tautologies. Smullyan [40, p. 51] extends his notion of a *Boolean valuation* from propositional logic to the syntax of first-order logic by treating quantified formulas as another sort of propositional symbols. A *tautology* is then a formula that is true under all Boolean valuations.

The following definition uses the same principle, where G is a predicate denotation as before and A is a special “universally quantified formula denotation.”

primrec boolean where
 $boolean \ - \ \perp = False$
 $| \ boolean \ G \ - \ (\dagger P \ ts) = G \ P \ ts$
 $| \ boolean \ G \ A \ (p \ \longrightarrow \ q) = (boolean \ G \ A \ p \ \longrightarrow \ boolean \ G \ A \ q)$
 $| \ boolean \ - \ A \ (\forall p) = A \ (\forall p)$

The hyphens stand for ignored arguments. Compare this semantics to the first-order one: it is indeed a Boolean valuation [40] of first-order logic. We can now take Smullyan’s notion of tautology as definition:

abbreviation $tautology \ p \equiv \forall \ G \ A. \ boolean \ G \ A \ p$

Smullyan gives no details on his extension of Boolean valuations to first-order logic. The way I set it up, with a separate denotation for the quantified formulas, it can be directly related to the first-order semantics.

► **Lemma 7** (Boolean semantics). *The Boolean and first-order semantics coincide when G matches the first-order predicate semantics and A is the first-order semantics itself.*

■ $boolean \ (\lambda a. \ G \ a \ \circ \ map \ (E, \ F)) \ \llbracket E, \ F, \ G \rrbracket = \llbracket E, \ F, \ G \rrbracket$

Proof. By structural induction. ◀

► **Lemma 8** (Tautologies). *All tautologies are valid.*

■ $tautology \ p \implies \llbracket E, \ F, \ G \rrbracket \ p$

Proof. Since a tautology holds for any choice of G and A it holds in particular for those that coincide with the first-order semantics (cf. Lemma 7). ◀

For reassurance, Isabelle easily verifies that not all first-order validities are propositional tautologies (e.g. $(\forall x. P(x)) \longrightarrow P(a)$ is only the former):

proposition $\exists p. (\forall E \ F \ G. \llbracket E, \ F, \ G \rrbracket \ p) \wedge \neg \ tautology \ p$

5.5 The Inductively Defined Calculus

Finally, we are ready to define the calculus itself. I define it as an inductive predicate \vdash that holds exactly when a formula can be derived from the given axioms and rules. The previous work has made the definition simple:

inductive *Axiomatic* ($\vdash \ - \ [50] \ 50$) **where**
 $TA: \ tautology \ p \implies \vdash \ p$
 $| \ IA: \ \vdash \ \forall p \ \longrightarrow \ \langle t/0 \rangle p$
 $| \ MP: \ \vdash \ p \ \longrightarrow \ q \implies \vdash \ p \implies \vdash \ q$
 $| \ GR: \ \vdash \ q \ \longrightarrow \ \langle \star a/0 \rangle p \implies a \notin \ params \ \{p, \ q\} \implies \vdash \ q \ \longrightarrow \ \forall p$

The Tautology Axiom (*TA*) derives any tautology. The Instantiation Axiom (*IA*) states that a quantified formula implies its instantiation with any term. The Modus Ponens (*MP*) rule is stated as usual and lifts an implication between formulas to an implication between derivations. Finally, the Generalization Rule (*GR*) works under assumptions q and generalizes from an instance to a quantified formula, given that the witness (the constant) is fresh.

► **Theorem 9** (Soundness). *Any derivable formula is valid:*

$$\blacksquare \vdash p \implies \llbracket E, F, G \rrbracket p$$

Proof. By induction over the inductive definition of the axiomatic system for arbitrary function denotation F .

All cases except for *GR* can be proven automatically, with the case for *TA* relying on Lemma 8 about tautologies. In the *GR* case I apply the induction hypothesis not just once at plain F but at $F(a := x)$ for every element x of the domain:

$$\text{have } \llbracket E, F(a := x), G \rrbracket (q \longrightarrow \langle \star a / 0 \rangle p) \text{ for } x$$

This is enough help for Isabelle to prove the case. ◀

► **Corollary 10.** *Falsity cannot be derived:*

$$\blacksquare \neg (\vdash \perp)$$

5.5.1 Notation

For the proof of completeness I need to express that a formula can be derived from a set of *assumptions*. Instead of building this notion into the definition of the proof system, I am going to simulate it using chains of implications. The expression $[p_1, p_2, \dots, p_n] \rightsquigarrow q$ expands to $p_1 \longrightarrow p_2 \longrightarrow \dots \longrightarrow p_n \longrightarrow q$. It is defined by recursion on the list of assumptions:

$$\begin{array}{l} \text{primrec imply (infixr } \rightsquigarrow \text{ 56) where} \\ \quad ([] \rightsquigarrow q) = q \\ \quad | (p \# ps \rightsquigarrow q) = (p \longrightarrow ps \rightsquigarrow q) \end{array}$$

I then write $ps \vdash q$ to abbreviate $\vdash ps \rightsquigarrow q$:

When I talk about *assumptions* in a derivation I will always mean a finite list of formulas.

5.6 Derived Formulas

Due to my semantic characterization of the Tautology Axiom, the automation in Isabelle can easily prove that various propositional formulas (schemas) can be derived.

► **Lemma 11** (Derivations). *The S and K combinators, double negation elimination and contraposition in both directions can all be derived:*

$$\begin{array}{l} \blacksquare \vdash (p \longrightarrow q \longrightarrow r) \longrightarrow (p \longrightarrow q) \longrightarrow p \longrightarrow r \\ \blacksquare \vdash q \longrightarrow p \longrightarrow q \\ \blacksquare \vdash \neg \neg p \longrightarrow p \\ \blacksquare \vdash (p \longrightarrow q) \longrightarrow \neg q \longrightarrow \neg p \\ \blacksquare \vdash (\neg q \longrightarrow \neg p) \longrightarrow p \longrightarrow q \end{array}$$

Proof. By the Tautology Axiom. ◀

5.6.1 Generalization Rule

My use of chains of implications is a disadvantage to the GR rule since it works on the consequent but implication is right associative. Consider the following: we know that $ps \vdash \langle \star a/0 \rangle p$, for fresh a and want to use GR to derive $ps \vdash \forall p$. We can only do so if ps consists of exactly one formula q , as $ps \vdash p$ is short for $\vdash ps \rightsquigarrow q$. To circumvent this restriction, I derive the following variant of the rule.

► **Lemma 12** (GR' rule). *The following rule is derivable:*

$$GR': \vdash \neg \langle \star a/0 \rangle p \longrightarrow q \implies a \notin \text{params } \{p, q\} \implies \vdash \neg (\forall p) \longrightarrow q$$

Proof. Follows from the GR rule, modus ponens and the derivations in Lemma 11. ◀

Since this rule works on the left-hand side of the implication, the right-hand side can, without issues, be an arbitrarily long chain of implications. Smullyan [40, p. 83] himself uses this version of the rule in his System Q1' (but for notational reasons).

An alternative is to start from the existential quantifier, \exists , as primitive, rather than \forall , as the generalization rule for \exists works on the left-hand side of the implication [40]. However, it is less immediately clear why this rule for \exists can be called a generalization rule.

5.6.2 Working with Assumptions

The following is an assortment of useful lemmas for working with assumptions.

► **Lemma 13** (Assumptions). *The following are derivable: modus ponens under assumptions, derivation of any assumption, the deduction theorem in both directions, a cut rule, classical reasoning and finally a structural rule encompassing weakening, contraction and exchange:*

- $ps \vdash p \longrightarrow q \implies ps \vdash p \implies ps \vdash q$
- $p \in \text{set } ps \implies ps \vdash p$
- $ps \vdash p \longrightarrow q \implies p \# ps \vdash q$
- $p \# ps \vdash q \implies ps \vdash p \longrightarrow q$
- $p \# ps \vdash r \implies q \# ps \vdash p \implies q \# ps \vdash r$
- $(\neg p) \# ps \vdash \perp \implies ps \vdash p$
- $ps \vdash q \implies \text{set } ps \subseteq \text{set } ps' \implies ps' \vdash q$

Proof. By a mix of induction over the list of assumptions and propositional reasoning. ◀

6 Completeness

We are now ready to delve into the completeness proof itself. The plan is as follows. If we cannot derive a formula p under any assumptions from X then we cannot derive falsity from $\neg p$ and any assumptions from X either. Sets like $\{\neg p\} \cup X$ are *consistent* with respect to the proof system, as we cannot derive a contradiction from them. I formalize them in Section 6.1. These sets are defined based on the proof system but we will use them to build a model that contradicts the validity of p under X . For this purpose we must prove that two important types of formulas preserve consistency: fresh witnesses of existential formulas (*Henkin witnesses*) and instances of universal formulas.

Lindenbaum (according to Tarski [41]) showed how to extend a consistent set into a *maximal consistent set (MCS)*. Any proper superset of a maximal consistent set is inconsistent. In particular this means that for any formula p , an MCS contains exactly p or $\neg p$. Henkin [21],

showed the utility of adding the Henkin witnesses for existential formulas during Lindenbaum's construction. I formalize the construction and its consistency in Section 6.2 and prove that the result is maximal in Section 6.3.

The addition of Henkin witnesses ensures that our MCSs are *saturated*. Section 6.4 outlines the benefits of ensuring this by construction.

Instead of building a model directly from a maximal consistent saturated set, I introduce a standard layer of abstraction. In Section 6.5, I formalize the notion of a *Hintikka* set [40] using three simple conditions and prove a model existence theorem: given a Hintikka set H , I build a model from a Herbrand structure [10, 22] that satisfies exactly the formulas in H . I then prove that maximal consistent saturated sets are Hintikka sets.

In Section 6.6, I put all the pieces together. The model existence theorem gives us a model for $\neg p$ and all of X . Therefore, if p is in fact valid under assumptions from X , then it must be derivable or we have a contradiction.

6.1 Consistent Sets

The definition of consistency is straightforward. The set of formulas S is consistent when there is no list of assumptions S' , coming from S , that can be used to derive falsity:

definition *consistent* $S \equiv \nexists S'. \text{ set } S' \subseteq S \wedge S' \vdash \perp$

The following lemma will be useful.

► **Lemma 14** (Inconsistent addition). *Assume that S is consistent on its own but becomes inconsistent with the addition of a formula p . Then there exists a concrete list of assumptions S' , coming from S , such that $p \# S' \vdash \perp$:*

assumes *consistent* S **and** \neg *consistent* $(\{p\} \cup S)$
obtains S' **where** *set* $S' \subseteq S$ **and** $p \# S' \vdash \perp$

Proof. It follows from consistency and the structural lemma for assumptions (Lemma 13). ◀

It is important to prove that two types of formulas preserve consistency. The first type is fresh witnesses for existential formulas.

► **Lemma 15** (Consistency of fresh witnesses). *If a consistent set contains an existential formula $\neg(\forall p)$ then adding a witness $\neg\langle \star a/0 \rangle p$, for a fresh a , preserves consistency:*

assumes *consistent* S **and** $\neg(\forall p) \in S$ **and** $a \notin \text{params } S$
shows *consistent* $(\{\neg\langle \star a/0 \rangle p\} \cup S)$

Proof. We need to show that there is no finite subset from which we can derive falsity, so assume that indeed there is one. From Lemma 14 we can name the problematic assumptions:

then obtain S' **where** *set* $S' \subseteq S$ **and** $(\neg\langle \star a/0 \rangle p) \# S' \vdash \perp$

After showing that the side conditions are fulfilled, we can apply the GR' rule:

then have $\neg(\forall p) \# S' \vdash \perp$

But every assumption is now in S , which we assumed to be consistent, so we have reached the desired contradiction and proved the lemma. ◀

We shall also need that instantiating a universally quantified formula preserves consistency.

8:14 A Succinct Formalization of the Completeness of First-Order Logic

► **Lemma 16** (Consistency of instantiation). *If a consistent set contains a universal formula $\forall p$ then adding an instance $\langle t/0 \rangle p$, for any term t , preserves consistency:*

assumes *consistent* S **and** $\forall p \in S$
shows *consistent* $(\{\langle t/0 \rangle p\} \cup S)$

Proof. The proof proceeds as before and we start by naming the problematic assumptions from an assumed inconsistency (Lemma 14):

then obtain S' **where** *set* $S' \subseteq S$ **and** $\langle t/0 \rangle p \# S' \vdash \perp$

This time we make use of the Instantiation Axiom, instantiated to p and t :

moreover have $\vdash \forall p \longrightarrow \langle t/0 \rangle p$

With the deduction theorem, the cut rule and the structural lemma (Lemma 13), we can apply this implication to weaken the derivation of falsity:

ultimately have $\forall p \# S' \vdash \perp$

But again, these assumptions are all in S , which we assumed to be consistent, so this is a contradiction and adding the instance must also be consistent. ◀

6.2 Lindenbaum Extension

We turn now to a central construction. Note first that if the sets of variable, function and predicate symbols are countable, so too are the sets of terms and formulas (formalized in Section 6.6). Thus, we can enumerate the formulas as p_0, p_1, \dots and so on. Starting from a consistent set S_0 , which leaves infinitely many parameters unused, we then build a sequence of consistent sets in the following way. Given S_n , construct S_{n+1} as:

$$S_{n+1} = \begin{cases} w(*, p_n) \cup \{p_n\} \cup S_n & \text{if } \{p_n\} \cup S_n \text{ is consistent} \\ S_n & \text{otherwise} \end{cases}$$

where $*$ is the set of parameters in $\{p_n\} \cup S_n$.

The function w returns a singleton set with a fresh witness when p_n is an existential formula and the empty set otherwise. Usually, the availability of such fresh witnesses is guaranteed by extending the set of function symbols. I assume instead that the set of function symbols is infinite from the start and that S_0 leaves infinitely many parameters unused. I pass the parameters of $\{p_n\} \cup S_n$ to w . It can then pick a parameter that has not been used already. This is simpler than dealing with two sorts of function symbols.

In the Isabelle formalization, the enumeration of formulas is represented by a (surjective) function f from the set of natural numbers to the set of formulas (cf. Section 6.6). The expression $extend\ S\ f\ n$ constructs the set S_n starting from $S_0 = S$:

primrec *extend* **where**
 $extend\ S\ f\ 0 = S$
 $| extend\ S\ f\ (Suc\ n) =$
(let $S_n = extend\ S\ f\ n$ *in*
if *consistent* $(\{f\ n\} \cup S_n)$
then *witness* $(params\ (\{f\ n\} \cup S_n))\ (f\ n) \cup \{f\ n\} \cup S_n$
else $S_n)$

The function *witness* is simple:

fun *witness* **where**

witness *used* $(\neg (\forall p)) = \{\neg \langle \star(\text{SOME } a. a \notin \text{used})/0 \rangle p\}$
 | *witness* *- -* = $\{\}$

Its definition uses Hilbert's choice operator to pick a fresh parameter.

The maximal consistent set is given by taking the union of this sequence of sets: $\bigcup_{n \in \mathbb{N}} S_n$.
 In Isabelle, it becomes:

definition *Extend S f* $\equiv \bigcup n. \text{extend } S f n$

The following lemmas are needed later.

► **Lemma 17** (Lindenbaum bounds). *The starting set is included in its maximal extension and each set in the constructed sequence bounds the previous sets:*

- $S \subseteq \text{Extend } S f$
- $(\bigcup n \leq m. \text{extend } S f n) = \text{extend } S f m$

Proof. By definition and by induction on m , respectively. ◀

► **Lemma 18** (Lindenbaum parameters). *A witness includes only finitely many parameters and each set S_n contains finitely many more parameters than the starting set S_0 :*

- *finite* (*params* (*witness used* p))
- *finite* (*params* (*extend S f n*) – *params* S)

Proof. Since p contains finitely many parameters and by induction on n , respectively. ◀

6.2.1 Consistency

The consistency of each constructed set S_n is apparent.

► **Lemma 19** (Consistency of S_n). *When starting from a consistent S_0 with infinitely many unused parameters, any constructed S_n is consistent:*

assumes *consistent* S **and** *infinite* ($UNIV - \text{params } S$)
shows *consistent* (*extend S f n*)

Proof. By induction on n . The consistency of adding the witness follows from Lemma 15. The only complication is to prove that there are indeed always fresh parameters available and therefore that the parameter given by Hilbert's choice operator is usable, but this follows from Lemma 18. ◀

The consistency of the union $\bigcup_n S_n$ is more interesting.

► **Lemma 20** (Consistency of $\bigcup_n S_n$). *The maximal extension of a consistent set S with infinitely many unused parameters is consistent:*

assumes *consistent* S **and** *infinite* ($UNIV - \text{params } S$)
shows *consistent* (*Extend S f*)

Proof. Assume towards a contradiction that we can derive falsity from some finite subset:

then obtain S' **where** $S' \vdash \perp$ **and** *set* $S' \subseteq \text{Extend } S f$

Since this subset is finite, it must be a subset of some initial segment of the union:

then obtain m **where** *set* $S' \subseteq (\bigcup n \leq m. \text{extend } S f n)$

But, by Lemma 17, each such segment is bounded by its last element:

then have *set* $S' \subseteq \text{extend } S f m$

And since we have already shown the consistency of each S_n (Lemma 19), we reach our desired contradiction. ◀

6.3 Maximal Sets

A maximal set is inconsistent under any proper extension:

definition *maximal* $S \equiv \forall p. p \notin S \longrightarrow \neg \text{consistent} (\{p\} \cup S)$

Maximal consistent sets are truly maximal:

► **Lemma 21** (Maximality of Maximal Consistent Sets). *If S is a maximal consistent set, then for every formula p , $p \in S$ if and only if $\neg p \notin S$.*

assumes *consistent* S **and** *maximal* S
shows $p \in S \longleftrightarrow (\neg p) \notin S$

Proof. The left-to-right direction follows from consistency alone and the right-to-left direction follows from consistency and maximality. ◀

That the Lindenbaum extension results in a maximal set is very easy to see.

► **Lemma 22** (Maximality of $\bigcup_n S_n$). *Given a surjective enumeration f , $\bigcup_n S_n$ is maximal:*

assumes *surj* f
shows *maximal* (*Extend* $S f$)

Proof. Assume towards a contradiction that some formula p is not included even though its inclusion preserves consistency:

assume $p \notin \text{Extend } S f$ **and** *consistent* ($\{p\} \cup \text{Extend } S f$)

Say that p is formula number k in the enumeration. Since p is not in the result, it must be inconsistent with S_k :

then have $\neg \text{consistent} (\{p\} \cup \text{extend } S f k)$

And this set is a subset of the final result:

moreover have $\{p\} \cup \text{extend } S f k \subseteq \{p\} \cup \text{Extend } S f$

Ultimately, this contradicts the assumption that adding p preserves consistency. ◀

6.4 Saturation

We shall need saturation to show that our constructed sets are Hintikka sets:

definition *saturated* $S \equiv \forall p. \neg (\forall p) \in S \longrightarrow (\exists a. (\neg \langle \star a / 0 \rangle p) \in S)$

So, in a saturated set there is a corresponding Henkin witness for each existential formula.

► **Lemma 23** (Saturation of $\bigcup_n S_n$). *A consistent Lindenbaum extension is saturated:*

assumes *consistent* (*Extend* $S f$) **and** *surj* f
shows *saturated* (*Extend* $S f$)

Proof. Guaranteed by construction. ◀

If we only constructed our set to be maximal consistent and tried to show that it was also saturated, we would run into trouble [40, p. 96]. First, given an arbitrary maximal consistent set S , it might be that a Henkin witness is missing because S includes every parameter available and every reuse of a parameter results in an inconsistency. Second, we might be unlucky and enumerate the negation of every suitable witness before enumerating the witness itself: we might always add the negation and never the witness. Following Henkin [21], I ensure saturation by adding the Henkin witnesses together with the existential formulas.

6.5 Hintikka Sets

Instead of showing the model existence theorem directly for maximal consistent saturated sets, it will be cleaner to show that Hintikka sets induce a model for their formulas and that our sets are in fact Hintikka sets.

The following definition characterizes a Hintikka set H over our syntax:

locale *Hintikka* =
fixes $H :: ('f, 'p)$ *fm set*
assumes
 $FlsH: \perp \notin H$ **and**
 $ImpH: (p \longrightarrow q) \in H \longleftrightarrow (p \in H \longrightarrow q \in H)$ **and**
 $UniH: (\forall p \in H) \longleftrightarrow (\forall t. \langle t/0 \rangle p \in H)$

Hintikka sets are sets that are *saturated downwards* [40, p. 27] and induce a model for the formulas in them. Since the set should induce a model, \perp should never be present ($FlsH$). Following Forster et al. [11, Lemma 11], I enforce that the set *respects* both implication ($ImpH$) and universal quantification ($UniH$): a formula is in the Hintikka set if and only if the “evidence” for that formula is also present. Here, evidence is to be understood in terms of the Herbrand model given below.

6.5.1 Model Existence

The model induced by a Hintikka set H is very simple. It consists of a Herbrand structure [10] and a predicate denotation based on H itself:

Domain Herbrand universe: the universe of terms.

Function denotation The constructor \dagger , i.e. every function symbol evaluates to itself.

Predicate denotation Predicate P is true for terms ts exactly when $\dagger P \ ts \in H$.

Like in the work by Herbelin and Ilik [22], but unlike for instance the formalizations by Berghofer [3] and Forster et al. [11], the Herbrand universe includes all terms, not just those with no variables. I never formalize what it means for a formula to be closed. The Herbrand structure famously evaluates any term without variables to itself [10]. Or in this case:

► **Lemma 24** (Herbrand semantics). *Under any Herbrand structure and the specific environment $\#$, every term evaluates to itself:*

■ $(\#, \dagger) \ t = t$

Proof. By structural induction. ◀

I reuse the notation for semantics and abbreviate the model induced by H as $\llbracket H \rrbracket$:

abbreviation $hmodel \ (\llbracket - \rrbracket)$ **where** $\llbracket H \rrbracket \equiv \llbracket \#, \dagger, \lambda P \ ts. \dagger P \ ts \in H \rrbracket$

We now reach the model existence theorem.

► **Theorem 25** (Model existence). *When H is a Hintikka set, $\llbracket H \rrbracket$ satisfies exactly the formulas in H .*

assumes *Hintikka* H
shows $p \in H \longleftrightarrow \llbracket H \rrbracket \ p$

Proof. By well-founded induction on the size of the formula as given by *size-fm*. Thus the induction hypothesis applies to any formula that is smaller by this measure, i.e. to subformulas and to instances of universally quantified formulas (cf. Lemma 6). These are exactly the

formulas that appear in the Hintikka conditions. The proof proceeds by considering each type of formula. Since there is a Hintikka condition for every type, which corresponds exactly to the semantics of the induced model, Isabelle automatically proves each case. For instance, a universal formula p is in the Hintikka set iff every instance $\langle t/0 \rangle p$ is in the Hintikka set (*UniH*) iff every instance $\langle t/0 \rangle p$ holds in the induced model (by the induction hypothesis). ◀

6.5.2 Saturated MCSs are Hintikka Sets

Consider first the following correspondence between derivability and MCSs.

► **Lemma 26** (Derivability and MCSs). *A formula p is derivable from an MCS S iff p is in S :*

assumes *consistent S and maximal S*
shows $(\exists ps. \text{set } ps \subseteq S \wedge ps \vdash p) \longleftrightarrow p \in S$

Proof. The left to right direction follows from the maximality of MCSs. The right to left direction follows trivially from the derivability of any assumption (Lemma 13). ◀

I now show that maximal consistent saturated sets are Hintikka sets.

► **Lemma 27** (Saturated MCSs are Hintikka sets). *If the set H is consistent, maximal and saturated, it is a Hintikka set:*

assumes *consistent H and maximal H and saturated H*
shows *Hintikka H*

Proof. We need to prove each case of the Hintikka definition. Take first the *FalsH* case:

show $\perp \notin H$

We need to show that falsity does not appear in our set. This follows directly from Lemma 26 and the assumed consistency of H .

Consider next the *ImpH* case:

show $(p \longrightarrow q) \in H \longleftrightarrow (p \in H \longrightarrow q \in H)$

From left to right, by using Lemma 26 this simply becomes modus ponens: if both $p \longrightarrow q$ and p are derivable from H then q must be derivable from H . The right to left direction is similar. It relies on Lemma 26, contraposition and Lemma 21: that exactly one of a formula and its negation is present in an MCS.

Consider next the *UniH* case:

show $(\forall p \in H) \longleftrightarrow (\forall t. \langle t/0 \rangle p \in H)$

One direction follows directly from consistency of instantiation (Lemma 16) and the maximality of H . The other direction follows from saturation (and Lemma 21). ◀

6.6 Completeness Theorem

Isabelle can automatically prove the countability of our syntax:

instance *tm* :: (countable) countable
instance *fm* :: (countable, countable) countable

These commands provide instances of the surjective function *from-nat* that takes natural numbers and returns terms and formulas, respectively. I state the main theorem as follows.

► **Theorem 28** (Completeness). *Assume that formula p is valid under assumptions X and that X leaves infinitely many parameters unused. Then we can derive p from X .*

fixes $p :: ('f :: \text{countable}, 'p :: \text{countable}) \text{ fm}$
assumes $\forall (E :: - \Rightarrow 'f \text{ tm}) F G. (\forall q \in X. \llbracket E, F, G \rrbracket q) \longrightarrow \llbracket E, F, G \rrbracket p$
and $\text{infinite } (UNIV - \text{params } X)$
shows $\exists ps. \text{set } ps \subseteq X \wedge ps \vdash p$

Proof. By contradiction:

assume $\nexists ps. \text{set } ps \subseteq X \wedge ps \vdash p$
then have $*$: $\nexists ps. \text{set } ps \subseteq X \wedge ((\neg p) \# ps \vdash \perp)$

If no such list of assumptions exists, then (by classical reasoning on the object level) there is also no list that allows us to derive falsity when assuming $\neg p$.

I introduce some local abbreviations $?S$ and $?H$ (where $?$ is required by Isabelle):

let $?S = \{\neg p\} \cup X$
let $?H = \text{Extend } ?S \text{ from-nat}$

It is easy to see from $*$ above that $?S$ must be consistent and the extension $?H$ is therefore maximal consistent (Lemmas 20, 22):

have $\text{consistent } ?S$
moreover have $\text{infinite } (UNIV - \text{params } ?S)$
ultimately have $\text{consistent } ?H$ **and** $\text{maximal } ?H$

$?H$ is saturated (Lemma 23) and Hintikka (Lemma 27):

moreover from $\text{this have saturated } ?H$
ultimately have $\text{Hintikka } ?H$

The model induced by $?H$ satisfies any formula in $?H$ (Theorem 25), including the starting set $?S$ (Lemma 17):

have $\llbracket ?H \rrbracket p$ **if** $p \in ?S$ **for** p
then have $\llbracket ?H \rrbracket (\neg p)$ **and** $\forall q \in X. \llbracket ?H \rrbracket q$

But this includes all formulas in X so by the assumed validity, $\llbracket H \rrbracket$ must also satisfy p and we reach our contradiction:

moreover from $\text{this have } \llbracket ?H \rrbracket p$
ultimately show False

The proof system is complete. ◀

The following abbreviation of validity in a specific Herbrand universe, with countably infinite function and predicate symbols, makes the result simpler to state:

abbreviation $\text{valid} :: (\text{nat}, \text{nat}) \text{ fm} \Rightarrow \text{bool}$ **where**
 $\text{valid } p \equiv \forall (E :: \text{nat} \Rightarrow \text{nat } \text{tm}) F G. \llbracket E, F, G \rrbracket p$

I fix the function and predicate symbols to be natural numbers but any countably infinite type works. One thing to note is that I only assume validity in one domain (the Herbrand universe), as I cannot quantify over the type I use to represent the domain. This is, however, a weaker assumption than assuming validity in all domains as is usually done.

► **Theorem 29** (Soundness and completeness). *Exactly the valid formulas are derivable:*

theorem main: $\text{valid } p \longleftrightarrow (\vdash p)$

Proof. By Theorems 9, 28. ◀

Only the definitions in Sections 3, 4 and Sections 5.1 to 5.5 must be inspected to trust the result. The definitions in this Section are only used for the proof.

7 Discussion

There are many choices to make in a formalization like this one. I choose to work with de Bruijn indices rather than named variables or Nominal Isabelle [42], which provides automation for this situation. While this choice makes it more complicated to explain the formalizations of e.g. semantics and quantifier instantiation, it makes the formalization self-contained. I hope to have demonstrated that the definitions themselves are simple, the functions are short and only a few simple lemmas are needed about them.

Recall the *GR* rule which is used in Lemma 15 to justify the consistency of fresh witnesses:

$$GR: \vdash q \longrightarrow \langle \star a/0 \rangle p \implies a \notin \text{params } \{p, q\} \implies \vdash q \longrightarrow \forall p$$

Since I use de Bruijn indices, this could also be formalized without the use of a parameter a by *lifting* q , in the sense of \uparrow , to ensure that variable 0 in p is safe to generalize directly:

$$\vdash \uparrow q \rightarrow p \implies \vdash q \rightarrow \forall p$$

However, we would then need to ensure that the entire set S' in Lemma 15 is *lifted* in order to apply the rule. With the present *GR* rule, we simply ensure that a is chosen to be fresh. It would be interesting to try Laurent's anti-locally nameless approach to quantifiers [27] and see whether this would yield a simpler formalization.

Another choice has been to simulate assumptions in derivations by a chain of implications. This trick applies directly to a one-sided calculus and makes it a lot simpler to work with, especially with some custom notation. It works especially well with Smullyan's System Q1 where the generalization rule (*GR*) works under an implication. The semantic characterization of the tautology axiom, which works well with Isabelle's proof automation, makes it even smoother since propositional reasoning becomes a non-issue.

One challenge was the realization that the variant *GR'* is more suitable than *GR*. Isabelle cannot tell us something like this, nor is the proof automation powerful enough to derive the rule automatically. The insight comes from experimenting with the formalization and proofs.

Some of these issues could also be resolved by starting from a natural deduction system rather than Smullyan's Hilbert system. Natural deduction systems have a context built in, where I must simulate it with implications, and more *natural* rules for the connectives, which could be used instead of the semantic characterization of tautologies. It remains future work to adapt the formalization to this setting and review the potential benefits.

At this point in time there is a large body of formalizations to draw on. I am inspired by Berghofer's formalization [3] of the completeness of natural deduction for first-order logic. Berghofer also formalizes Lindenbaum's construction and my definition is close to his. My formalization of Hintikka sets and the model existence theorem, however, is both shorter (due to Forster et al. [11]) and, unlike Berghofer's, works directly for open formulas (cf. Herbelin and Ilik [22]). As such, even though some notion has already been formalized, it can be beneficial to revisit it.

8 Conclusion and Future Work

I have used techniques from computer science like de Bruijn indices and functional programming to work in the meta-logic of the proof assistant Isabelle/HOL. Here, I have formalized the syntax and semantics of first-order logic and defined a simple axiomatic proof system for it. This definition has included careful considerations of the interplay between syntax and semantics, a semantic characterization of tautologies suitable for formalization and notational tricks like the use of implications to simulate assumptions.

I have then carried out a completeness proof for the Hilbert system in the style of Henkin, and using ideas from Lindenbaum, Hintikka and Herbrand along the way. The proof is direct: use Lindenbaum’s construction to extend a consistent set to a maximal consistent set, add Henkin witnesses of existential formulas during this construction, notice that the result is a Hintikka set and build a model in the Herbrand universe. Section 2 demonstrated the usefulness of this style in the formalization of other logics and proof systems. My formalization may serve as starting point for such endeavors: researchers can modify the existing definitions and proofs rather than start from scratch. Isabelle/HOL ensures that such modifications are correct and can help fill in gaps in the proofs when they arise. This provides an entry point to formalizing such a completeness proof.

In the future, however, I want to abstract this proof along several dimensions. First, the entire construction outlined above could potentially be given in the abstract and instantiated with a concrete proof system, witnessing function, notion of saturation, etc. Then it might be shared among the several formalizations of this method, and potential new ones. Popescu and Traytel [36] have already developed some syntax-independent logical infrastructure in their formal verification of an abstract account of Gödel’s incompleteness theorems. This future work could potentially build on theirs, extending it with the model existence theorem and more. Second, Smullyan gives many constructions in his uniform notation that abstracts over the concrete choice of syntax. I would like to abstract this formalization in a similar way: witnesses could be added for “ δ -formulas”, which might happen to be of the form $\neg (\forall p)$ like here or maybe of the form $\diamond p$ as seen in hybrid logic [26].

I also want to extend the syntax, semantics, proof system and completeness proof to first-order logic with equality. I already handle function symbols, unlike Smullyan, but to get on par with Barwise, equality needs to be considered too. The Henkin style should scale well for this extension. The current formalization does, however, have the benefit of outlining the fundamental ideas of the completeness proof without too many auxiliary considerations. This is an advantage for adapting it to other logics.

I hope this formalization will serve as inspiration, and perhaps as a starting point, for further formalizations of logic.

References

- 1 Jon Barwise. An introduction to first-order logic. In Jon Barwise, editor, *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, pages 5–46. Elsevier, 1977. doi:10.1016/S0049-237X(08)71097-8.
- 2 Bruno Bentzen. A Henkin-style completeness proof for the modal logic S5. In Pietro Baroni, Christoph Benzmüller, and Yi N. Wáng, editors, *Logic and Argumentation - 4th International Conference, CLAR 2021, Hangzhou, China, October 20-22, 2021, Proceedings*, volume 13040 of *Lecture Notes in Computer Science*, pages 459–467. Springer, 2021. doi:10.1007/978-3-030-89391-0_25.
- 3 Stefan Berghofer. First-order logic according to Fitting. *Archive of Formal Proofs*, August 2007. Formal proof development. URL: <https://isa-afp.org/entries/FOL-Fitting.html>.
- 4 Jasmin Christian Blanchette. Formalizing the metatheory of logical calculi and automatic provers in Isabelle/HOL (invited talk). In Assia Mahboubi and Magnus O. Myreen, editors, *Proceedings of the 8th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2019*, pages 1–13. ACM, 2019.
- 5 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Abstract completeness. *Archive of Formal Proofs*, April 2014. Formal proof development. URL: https://isa-afp.org/entries/Abstract_Completeness.html.

- 6 Jasmin Christian Blanchette, Andrei Popescu, and Dmitriy Traytel. Soundness and completeness proofs by coinductive methods. *Journal of Automated Reasoning*, 58(1):149–179, 2017. doi:10.1007/s10817-016-9391-3.
- 7 Patrick Braselmann and Peter Koepke. Gödel’s completeness theorem. *Formalized Mathematics*, 13(1):49–53, 2005.
- 8 Robert L. Constable and Mark Bickford. Intuitionistic completeness of first-order logic. *Annals of Pure and Applied Logic*, 165(1):164–198, 2014. doi:10.1016/j.apal.2013.07.009.
- 9 N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. In R.P. Nederpelt, J.H. Geuvers, and R.C. de Vrijer, editors, *Selected Papers on Automath*, volume 133 of *Studies in Logic and the Foundations of Mathematics*, pages 375–388. Elsevier, 1994. Reprinted from: *Indagationes Math*, 34, 5, p. 381-392, by courtesy of the Koninklijke Nederlandse Akademie van Wetenschappen, Amsterdam. doi:10.1016/S0049-237X(08)70216-7.
- 10 Melvin Fitting. *First-Order Logic and Automated Theorem Proving, Second Edition*. Graduate Texts in Computer Science. Springer, 1996.
- 11 Yannick Forster, Dominik Kirst, and Dominik Wehr. Completeness theorems for first-order logic analysed in constructive type theory. *Journal of Logic and Computation*, 31(1):112–151, 2021. doi:10.1093/logcom/exaa073.
- 12 Asta Halkjær From. Synthetic completeness for a terminating Seligman-style tableau system. 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 5:1–5:17. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020. doi:10.4230/LIPICs.TYPES.2020.5.
- 13 Asta Halkjær From. Formalizing Henkin-style completeness of an axiomatic system for propositional logic. In *WESSLLI + ESSLLI Virtual Student Session*, 2021. Accepted for post-proceedings.
- 14 Asta Halkjær From, Anders Schlichtkrull, and Jørgen Villadsen. A sequent calculus for first-order logic formalized in Isabelle/HOL. In Stefania Monica and Federico Bergenti, editors, *Proceedings of the 36th Italian Conference on Computational Logic, Parma, Italy, September 7-9, 2021*, volume 3002 of *CEUR Workshop Proceedings*, pages 107–121. CEUR-WS.org, 2021. URL: <http://ceur-ws.org/Vol-3002/paper7.pdf>.
- 15 Asta Halkjær From. Formalizing a Seligman-style tableau system for hybrid logic. *Archive of Formal Proofs*, December 2019. Formal proof development. URL: http://isa-afp.org/entries/Hybrid_Logic.html.
- 16 Asta Halkjær From. A sequent calculus for first-order logic. *Archive of Formal Proofs*, July 2019. Formal proof development. URL: https://isa-afp.org/entries/FOL_Seq_Calc1.html.
- 17 Asta Halkjær From. Soundness and completeness of an axiomatic system for first-order logic. *Archive of Formal Proofs*, September 2021. Formal proof development. URL: https://isa-afp.org/entries/FOL_Axiomatic.html.
- 18 Herman Geuvers. Proof assistants: History, ideas and future. *Sadhana*, 34(1):3–25, 2009. doi:10.1007/s12046-009-0001-5.
- 19 Kurt Gödel. Die Vollständigkeit der Axiome des logischen Funktionenkalküls. *Monatshefte für Mathematik und Physik*, 37(1):349–360, 1930.
- 20 John Harrison. Formalizing basic first order model theory. In Jim Grundy and Malcolm C. Newey, editors, *Theorem Proving in Higher Order Logics, 11th International Conference, TPHOLs’98, Canberra, Australia, September 27 - October 1, 1998, Proceedings*, volume 1479 of *Lecture Notes in Computer Science*, pages 153–170. Springer, 1998. doi:10.1007/BFb0055135.
- 21 Leon Henkin. The discovery of my completeness proofs. *Bulletin of Symbolic Logic*, 2(2):127–158, 1996. doi:10.2307/421107.
- 22 Hugo Herbelin and Danko Ilik. An analysis of the constructive content of Henkin’s proof of Gödel’s completeness theorem. *Manuscript available online*, 2016.

- 23 Jacques Herbrand. *Recherches sur la théorie de la démonstration*. Number 110 in Thèses de l'entre-deux-guerres. Faculté des Sciences de L'Université de Paris, 1930.
- 24 Denis R. Hirschfeldt. *Slicing the Truth - On the Computable and Reverse Mathematics of Combinatorial Principles*, volume 28 of *Lecture Notes Series / Institute for Mathematical Sciences / National University of Singapore*. World Scientific, 2014. doi:10.1142/9208.
- 25 Danko Ilik. *Constructive Completeness Proofs and Delimited Control. (Preuves constructives de complétude et contrôle délimité)*. PhD thesis, École Polytechnique, Palaiseau, France, 2010. URL: <https://tel.archives-ouvertes.fr/tel-00529021>.
- 26 Klaus Frovin Jørgensen, Patrick Blackburn, Thomas Bolander, and Torben Braüner. Synthetic completeness proofs for Seligman-style tableau systems. In Lev D. Beklemishev, Stéphane Demri, and András Maté, editors, *Advances in Modal Logic 11, proceedings of the 11th conference on "Advances in Modal Logic," held in Budapest, Hungary, August 30 - September 2, 2016*, pages 302–321. College Publications, 2016. URL: <http://www.aiml.net/volumes/volume11/Joergensen-Blackburn-Bolander-Brauner.pdf>.
- 27 Olivier Laurent. An anti-locally-nameless approach to formalizing quantifiers. In Catalin Hritcu and Andrei Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 300–312. ACM, 2021. doi:10.1145/3437992.3439926.
- 28 Pierre Lescanne. From lambda-sigma to lambda-epsilon: a journey through calculi of explicit substitutions. In Hans-Juergen Boehm, Bernard Lang, and Daniel M. Yellin, editors, *Conference Record of POPL'94: 21st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Portland, Oregon, USA, January 17-21, 1994*, pages 60–69. ACM Press, 1994. doi:10.1145/174675.174707.
- 29 James Margetson and Tom Ridge. Completeness theorem. *Archive of Formal Proofs*, September 2004. Formal proof development. URL: <http://isa-afp.org/entries/Completeness.html>.
- 30 Julius Michaelis and Tobias Nipkow. Propositional proof systems. *Archive of Formal Proofs*, June 2017. Formal proof development. URL: http://isa-afp.org/entries/Propositional_Proof_Systems.html.
- 31 Julius Michaelis and Tobias Nipkow. Formalized Proof Systems for Propositional Logic. In Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi, editors, *23rd International Conference on Types for Proofs and Programs (TYPES 2017)*, volume 104 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:16, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2017.5.
- 32 Tobias Nipkow. More Church-Rosser proofs. *Journal of Automated Reasoning*, 26(1):51–66, 2001. doi:10.1023/A:1006496715975.
- 33 Tobias Nipkow and Gerwin Klein. *Concrete Semantics - With Isabelle/HOL*. Springer, 2014. doi:10.1007/978-3-319-10542-0.
- 34 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002. doi:10.1007/3-540-45949-9.
- 35 Henrik Persson. Constructive completeness of intuitionistic predicate logic. *Licenciate thesis, Chalmers University of Technology*, 1996.
- 36 Andrei Popescu and Dmitriy Traytel. A formally verified abstract account of Gödel's incompleteness theorems. In Pascal Fontaine, editor, *Automated Deduction - CADE 27 - 27th International Conference on Automated Deduction, Natal, Brazil, August 27-30, 2019, Proceedings*, volume 11716 of *Lecture Notes in Computer Science*, pages 442–461. Springer, 2019. doi:10.1007/978-3-030-29436-6_26.
- 37 Anders Schlichtkrull. The resolution calculus for first-order logic. *Archive of Formal Proofs*, June 2016. Formal proof development. URL: http://isa-afp.org/entries/Resolution_FOL.html.
- 38 Anders Schlichtkrull. Formalization of the resolution calculus for first-order logic. *Journal of Automated Reasoning*, 61(1-4):455–484, 2018. doi:10.1007/s10817-017-9447-z.

8:24 A Succinct Formalization of the Completeness of First-Order Logic

- 39 Natarajan Shankar. Towards mechanical metamathematics. *Journal of Automated Reasoning*, 1(4):407–434, 1985.
- 40 Raymond M. Smullyan. *First-Order Logic*. Springer-Verlag, 1968.
- 41 Alfred Tarski. *Logic, Semantics, Metamathematics: Papers from 1923 to 1938*. Hackett Publishing, 1983.
- 42 Christian Urban. Nominal techniques in Isabelle/HOL. *Journal of Automated Reasoning*, 40(4):327–356, 2008. doi:10.1007/s10817-008-9097-2.
- 43 Wim Veldman. An intuitionistic completeness theorem for intuitionistic predicate logic. *Journal of Symbolic Logic*, 41(1):159–166, 1976. doi:10.1017/S0022481200051859.

Simulating Large Eliminations in Cedille

Christa Jenkins ✉ 🏠 

The University of Iowa, Iowa City, IA, USA

Andrew Marmaduke ✉ 🏠

The University of Iowa, Iowa City, IA, USA

Aaron Stump ✉ 🏠

The University of Iowa, Iowa City, IA, USA

Abstract

Large eliminations provide an expressive mechanism for arity- and type-generic programming. However, as large eliminations are closely tied to a type theory's primitive notion of inductive type, this expressivity is not expected within polymorphic lambda calculi in which datatypes are encoded using impredicative quantification. We report progress on simulating large eliminations for datatype encodings in one such type theory, the *calculus of dependent lambda eliminations* (CDLE). Specifically, we show that the expected computation rules for large eliminations, expressed using a derived type of extensional equality of types, can be proven *within* CDLE. We present several case studies, demonstrating the adequacy of this simulation for a variety of generic programming tasks, and a generic formulation of the simulation allowing its use for a broad family of datatype encodings. All results have been mechanically checked by Cedille, an implementation of CDLE.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases large eliminations, generic programming, impredicative encodings, Cedille, Mendler algebra

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.9

Supplementary Material *Software (Source Code):*

<https://github.com/cedille/cedille-developments/>

archived at `swh:1:dir:8b0dbbbb5203be35a7242a469f45a9cdbffcebfa`

1 Introduction

In dependently typed languages, large eliminations allow programmers to define types by induction over datatypes – that is, as an elimination of a datatype into the large universe of types. For type theory semanticists, large eliminations rule out two-element models of types by providing a principle of proof discrimination (e.g., $0 \neq 1$) [26, 25]. For programmers, they give an expressive mechanism for arity- and type-generic programming with universe constructions [34]. As an example, the type *Nary* n of n -ary functions (where n is a natural number) over type T can be defined as T when $n = 0$ and $T \rightarrow \text{Nary } n'$ when $n = \text{succ } n'$.

Large eliminations are closely tied to a type theory's primitive notion of inductive type. Thus, this expressivity is not expected within polymorphic pure typed lambda calculi in which datatypes are impredicatively encoded. The *calculus of dependent lambda eliminations* (CDLE) [27, 28] is one such theory that seeks to overcome historical difficulties of impredicative encodings, such as the lack of induction principles for datatypes [13].

Contributions. In this paper, we report progress on overcoming another difficulty of impredicative encodings: the lack of large eliminations. We show that the expected definitional equalities of a large elimination can be simulated using a *derived type of extensional equality* for types (as CDLE is an extrinsic theory, we take the extent of a type to be the set of terms it classifies). In particular, we:

- describe our method for simulating large eliminations in CDLE (Section 3) using a concrete example, identifying the features of the theory that enable the development (Section 2) and noting a limitation on the contexts in which it may be effectively used;



© Christa Jenkins, Andrew Marmaduke, and Aaron Stump;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 9; pp. 9:1–9:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

- formulate the method *generically* for all impredicative encodings of the form used by the datatype system of the Cedille tool (Section 5);
- demonstrate the adequacy of this simulation by applying it to several generic programming tasks: n -ary functions, a closed universe of datatypes, and an arity-generic map operation (Sections 3 and 4).

All results have been mechanically checked by Cedille, an implementation of CDLE, and are available in the supplementary material for this paper.

Outline. Section 2 reviews background material on CDLE, focusing on the primitives which enable the simulation. In Section 3, we carefully explain the recipe for simulating large eliminations using as an example the type of n -ary functions over a given type. Section 4 shows two more case studies, a closed universe of strictly positive types and a generalized map operation for vectors, as evidence of the effectiveness of the simulation in tackling generic programming tasks. The recipe for concrete examples is then turned into a generic derivation (that is, parametric in a covariant datatype signature) of simulated large eliminations in Section 5. Finally, Section 6 discusses related work and Section 7 concludes with a discussion of future work.

2 Background on CDLE

In this section, we review CDLE, the kernel theory of Cedille. CDLE extends the impredicative extrinsically typed *calculus of constructions* (CC), overcoming historical difficulties of impredicative encodings (e.g., underivability of induction [14]) by adding three new type constructs: equality of untyped terms; the dependent intersections of Kopylov [20]; and the implicit products of Miquel [24]. The pure term language of CDLE is untyped lambda calculus, but to make type checking algorithmic terms t are presented with typing annotations which are removed during erasure (written $|t|$). Definitional equality of terms t_1 and t_2 is $\beta\eta$ -equivalence modulo erasure of annotations, denoted $|t_1| =_{\beta\eta} |t_2|$.

The typing and erasure rules for the fragment of CDLE used in this paper are shown in Figure 1 and described in Section 2.1 (see also Stump and Jenkins [28]); the derived constructs we use are presented axiomatically in Section 2.2. We assume the reader is familiar with the type constructs inherited from CC: abstraction over types in terms is written $\Lambda X. t$ (erasing to $|t|$), application of terms to types (polymorphic type instantiation) is written $t \cdot T$ (erasing to $|t|$), and application of type constructors to type constructors is written $T_1 \cdot T_2$. In code listings, we sometimes omit type arguments to terms when Cedille can infer them.

2.1 Primitives

Below, we only discuss implicit products and the equality type. Though dependent intersections play a critical role in the derivation of induction for datatype encodings, they are otherwise not explicitly used in the coming developments.

The implicit product type $\forall x:T_1. T_2$. The implicit product type $\forall x:T_1. T_2$ of Miquel [24] is the type for functions which accept an erased (computationally irrelevant) input of type T_1 and produce a result of type T_2 . Implicit products are introduced with $\Lambda x. t$, and the type inference rule is the same as for ordinary function abstractions except for the side condition that x does not occur free in the erasure of the body t . Thus, the argument plays no computational role in the function and exists solely for the purposes of typing: the erasure

$$\begin{array}{c}
\frac{\Gamma, x : T_1 \vdash t : T_2 \quad x \notin FV(|t|)}{\Gamma \vdash \Lambda x. t : \forall x : T_1. T_2} \quad \frac{\Gamma \vdash t : \forall x : T_1. T_2 \quad \Gamma \vdash t' : T_1}{\Gamma \vdash t - t' : [t'/x]T_2} \\
\\
\frac{|t_1| =_{\beta\eta} |t_2|}{\Gamma \vdash \beta : \{t_1 \simeq t_2\}} \quad \frac{\Gamma \vdash t : \{\lambda x. \lambda y. x \simeq \lambda x. \lambda y. y\}}{\Gamma \vdash \delta - t : T} \\
\\
\frac{\Gamma \vdash t : \{t' \simeq t''\} \quad \Gamma \vdash t' : T \quad FV(t'') \subseteq \text{dom}(\Gamma)}{\Gamma \vdash \varphi t - t' \{t''\} : T} \\
\\
\begin{array}{l}
|\Lambda x. t| = |t| \quad |t - t'| = |t| \\
|\beta| = \lambda x. x \quad |\varphi t - t' \{t''\}| = |t''| \\
|\delta - t| = \lambda x. x
\end{array}
\end{array}$$

■ **Figure 1** Typing and erasure for a fragment of CDLE.

of $\Lambda x. t$ is $|t|$. For application, if t has type $\forall x : T_1. T_2$ and t' has type T_1 , then $t - t'$ has type $[t'/x]T_2$ and erases to $|t|$. When x is not free in T_2 , we write $T_1 \Rightarrow T_2$, similar to writing $T_1 \rightarrow T_2$ for $\Pi x : T_1. T_2$.

► **Note.** The notion of computational irrelevance here is not that of a different sort of classifier for types (e.g. *Prop* in Coq [31]) separating terms by whether they can be used for computation. Instead, it is similar to *quantitative type theory* [2]: relevance and irrelevance are properties of *binders*, indicating how functions may use arguments.

The equality type $\{t_1 \simeq t_2\}$. The equality type $\{t_1 \simeq t_2\}$ is the type of proofs that t_1 is propositionally equal to t_2 . The introduction form β proves reflexive equations between $\beta\eta$ -equivalence classes of terms: it can be checked against the type $\{t_1 \simeq t_2\}$ if $|t_1| =_{\beta\eta} |t_2|$. Note that this means equality is over *untyped* (post-erasure) terms. There is also a standard elimination form (substitution), but it is not used explicitly in the presentation of our results, so we omit its inference rule.

Equality types also come with two additional axioms.

- The φ axiom gives a strong form of the direct computation rule of NuPRL (see Allen et al. [1], Section 2.2). Though φ does not appear explicitly in the developments to come, it plays a central role by enabling the derivation of extensional type equality that enables zero-cost coercions between the equated types.
- The δ axiom provides a principle of proof discrimination. By enabling proofs that datatype constructors are disjoint, δ plays a vital role in our simulation of large eliminations.

The inference rule for an expression of the form $\varphi t - t' \{t''\}$ says that the entire expression can be checked against type T if t' can be, if there are no undeclared free variables in t'' (so, t'' is a well-scoped but otherwise untyped term), and if t proves that t' and t'' are equal. The crucial feature of φ is its erasure: the expression erases to $|t''|$, effectively enabling us to cast t'' to the type of t' . An expression of the form $\delta - t$ may be checked against any type if t synthesizes a type convertible with a particular false equation, $\{\lambda x. \lambda y. x \simeq \lambda x. \lambda y. y\}$. To broaden applicability of δ , the Cedille tool implements the *Böhm-out* semi-decision procedure [4] for discriminating between separable lambda terms.

$$\frac{\Gamma \vdash t_1 : S \rightarrow T \quad \Gamma \vdash t_2 : \Pi x : S. \{t_1 x \simeq x\}}{\Gamma \vdash \text{intrCast} \cdot S \cdot T \text{ } -t_1 \text{ } -t_2 : \text{Cast} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \text{Cast} \cdot S \cdot T}{\Gamma \vdash \text{cast} \cdot S \cdot T \text{ } -t : S \rightarrow T} \quad \frac{\Gamma \vdash t : \text{Cast} \cdot S \cdot T}{\Gamma \vdash \text{etaCast} \cdot S \cdot T \text{ } -t : \{t \simeq \lambda x. x\}}$$

$$\begin{aligned} |\text{intrCast} \cdot S \cdot T \text{ } -t_1 \text{ } -t_2| &= \lambda x. x & |\text{cast} \cdot S \cdot T \text{ } -t| &= \lambda x. x \\ |\text{etaCast} \cdot S \cdot T \text{ } -t| &= |\lambda x. x| \end{aligned}$$

■ **Figure 2** Type inclusions.

$$\frac{\Gamma \vdash t_1 : \text{Cast} \cdot S \cdot T \quad \Gamma \vdash t_2 : \text{Cast} \cdot T \cdot S}{\Gamma \vdash \text{intrTpEq} \cdot S \cdot T \text{ } -t_1 \text{ } -t_2 : \text{TpEq} \cdot S \cdot T}$$

$$\frac{\Gamma \vdash t : \text{TpEq} \cdot S \cdot T}{\Gamma \vdash \text{tpEq1} \cdot S \cdot T \text{ } -t : S \rightarrow T} \quad \frac{\Gamma \vdash t : \text{TpEq} \cdot S \cdot T}{\Gamma \vdash \text{tpEq2} \cdot S \cdot T \text{ } -t : T \rightarrow S}$$

$$\begin{aligned} |\text{intrTpEq} \cdot S \cdot T \text{ } -t_1 \text{ } -t_2| &= \lambda x. x & |\text{tpEq1} \cdot S \cdot T \text{ } -t| &= \lambda x. x \\ |\text{tpEq2} \cdot S \cdot T \text{ } -t| &= \lambda x. x \end{aligned}$$

■ **Figure 3** Extensional type equality.

2.2 Derived Constructs

Type inclusions

The φ axiom of equality allows us to define a type constructor *Cast* that internalizes the notion that the set of all elements of some type S is contained within the set of all elements of type T (note that Curry-style typing makes this relation nontrivial). We describe its axiomatic summary, presented in Figure 2; for the full derivation, see Jenkins and Stump [17] (also Diehl et al. [11] for the related notion of Curry-style identity functions).

The introduction form *intrCast* takes two erased term arguments, a function $t_1 : S \rightarrow T$, and a proof that t_1 behaves extensionally as the identity function on its domain. The elimination form *cast* takes evidence that a type S is included into T and produces a function of type $S \rightarrow T$. The crucial property of *cast* is its erasure: $|\text{cast } -t| = \lambda x. x$. Thus, $\text{Cast} \cdot S \cdot T$ may also be considered the type of *zero-cost* type coercions from S to T – zero cost because the type coercion is performed in a constant number of β -reduction steps. The uniqueness principle *etaCast* tells us that every witness of a type inclusion is equal to $\lambda x. x$.

► **Note.** The significance to the results presented in this paper of the fact that any two witnesses of a type inclusion are equal is discussed in Remark 11.

► **Remark.** When inspecting the introduction and elimination forms, it may seem that *Cast* provides a form of function extensionality restricted to identity functions. This is not the case, however, as it is possible to choose S , T , and $t_1 : S \rightarrow T$ such that t_1 is provably extensionally equal to the identity function for terms of type S , and *at the same time* refute $\{t_1 \simeq \lambda x. x\}$ using δ . Instead, these rules should be read as saying that if t_1 is extensionally identity on its domain, then that fact justifies the *assignment of type* $S \rightarrow T$ to $\lambda x. x$.

Type equality

The extensional notion of type equality used to simulate large eliminations, $TPEq$, is the existence of a two-way type inclusion, as shown by the introduction form $intrTPEq$ in Figure 3. Similar to $Cast$, the important feature of the summary of $TPEq$ for the reader to keep in mind is the erasure rules for the elimination forms, $tpEq1$ and $tpEq2$: both erase to $\lambda x. x$. In Section 3.2, we will see a proof that computes a type equality witness in linear time. However, in both elimination forms the type equality witness $t : TPEq \cdot S \cdot T$ is given as an *erased* argument. This means that the complexity of computing the witness is irrelevant to the functions that realize the two-way coercion.

► **Remark.** Strictly speaking, the type $TPEq \cdot S \cdot T$ is defined as the intersection of the types $Cast \cdot S \cdot T$ and $Cast \cdot T \cdot S$. In particular, this means $TPEq$ enjoys the same uniqueness property as $Cast$ that all witnesses are equal to $\lambda x. x$. However, the developments of this paper do not need to make explicit use of this property, so we omit this from the figure.

2.2.1 Substitution

Though we call $TPEq$ extensional type *equality*, within CDLE it is only an isomorphism of types. To be considered a true notion of equality, $TPEq$ would need a substitution principle. The type constructors for dependent function types (both implicit and explicit) can be proven to permit substitution if the domain and codomain parts do, as does quantification over types. However, a proof of general substitution principle would assume an arbitrary type constructor $X : \star \rightarrow \star$ and a term $t : X \cdot S$, and would need to produce a term of type $X \cdot T$, where S and T are types such that $TPEq \cdot S \cdot T$. To proceed, we appear to require additional assumption on X – otherwise, we cannot decompose or analyze the type $X \cdot S$ any further.

Nonetheless, the case studies presented in Sections 3 and 4 show that despite this limitation, our simulation of large eliminations using $TPEq$ is adequate for dealing with common generic-programming tasks (see for example Note 9). Where we do use type constructors of higher order than \star (such as in Section 4.2.1), we restrict ourselves to those which admit a substitution principle for $TPEq$.

3 n -ary Functions

In this section, we use a concrete example to detail the method of simulating large eliminations. Figure 4a shows the definition of $Nary$, the family of n -ary function types over some type T , as a large elimination of natural numbers. Our simulation of this begins by approximating this inductive definition of a *function* with an inductive *relation* between Nat and types, given as the generalized algebraic datatype [36] (GADT) $NaryR$ in Figure 4b.

This approximation is inadequate: we lack a canonical name for the type $Nary\ n$ because n does not *a priori* determine the type argument of $NaryR\ n$. Indeed, without a form of proof discrimination we would not even be able to deduce that if a given type N satisfies

<pre>Nary : Nat → ★ Nary zero = T Nary (succ n) = T → Nary n</pre>	<pre>data NaryR : Nat → ★ → ★ = naryRZ : NaryR zero ·T naryRS : ∀ n: Nat. ∀ Ih: ★. NaryR n ·Ih → NaryR (succ n) ·(T → Ih)</pre>
--	---

(a) As a large elimination.

(b) As a GADT.

■ **Figure 4** n -ary functions over T [source].

$NaryR$ $zero$, then from a term of type N we can extract a term of type T . Proceeding by induction, in the $naryRS$ case the (impossible) goal is to show that T is the same as $T \rightarrow Ih$ for some arbitrary but fixed $Ih : \star$. We would need to derive a contradiction from the absurd equation that $\{succ\ n \simeq zero\}$ for some n . Fortunately, proof discrimination *is* available in CDLE in the form of δ , so we are able to define functions such as $extr0$ below which require this form of reasoning.

```
extr0' :  $\forall x : Nat. \{ x \simeq zero \} \Rightarrow \forall N : \star. NaryR\ x \cdot N \rightarrow N \rightarrow T$ 
extr0' -zero -eqx  $\cdot T\ naryRZ\ x = x$ 
extr0' -(succ n) -eqx  $\cdot (T \rightarrow X)\ (naryRS\ n \cdot X\ r)\ x = \delta - eqX$ 

extr0 = extr0' -zero - $\beta$ 
```

► **Note.** In code listings such as the above, we present recursive Cedille functions using the syntax of (dependent) pattern matching to aid readability. This syntax is not currently supported by the Cedille tool. For the functions we present, those that compute terms are implemented in this paper’s repository using the datatype system described by Jenkins et al. [16], and those that compute types use the simulation to be described next.

In the digital version of this paper, figures with code listings are accompanied by hyperlinks to the Cedille implementation embedded in the text “[source]” in captions.

3.1 Sketch of the Idea

Our task is to show that $NaryR$ defines a *functional* relation, i.e., for all $n : Nat$ there exists a unique type $Nary\ n$ such that $NaryR\ n \cdot (Nary\ n)$ is inhabited. The candidate definition for this type family is:

$$Nary\ n = \forall X : \star. NaryR\ n \cdot X \Rightarrow X$$

For all n , read $Nary\ n$ as the type of terms contained in the intersection of the family of types X such that $NaryR\ n \cdot X$ is inhabited. For example, every term t of type $Nary\ zero$ has type T also, since T is in this family (specifically, we have that $t \cdot T -naryRZ$ has type T and erases to $|t|$). In the other direction, every term of type T also has type $Nary\ zero$, since the only type X satisfying $NaryR\ zero \cdot X$ is T itself.

However, at the moment we are stuck when attempting to prove $NaryR\ zero \cdot (Nary\ zero)$. Though we see from the preceding discussion that T and $Nary\ zero$ are *extensionally* equal types (they classify the same terms), $naryRZ$ requires that they be *definitionally* equal! Furthermore, and as noted in Section 2.2.1, derived extensional type equality does not admit a general substitution principle, which would allow us to rewrite the type $NaryR\ zero \cdot T$ to the desired type by proving $TpEq \cdot T \cdot (Nary\ zero)$. Therefore, we must modify the definition of $NaryR$ so that it defines a relation that is functional with respect to extensional type equality. This is shown in below, with both constructors now quantifying over an additional type argument X together with evidence that it is extensionally equal to the type of interest.

```
data NaryR : Nat  $\rightarrow \star \rightarrow \star$ 
= naryRZ :  $\forall X : \star. TpEq \cdot X \cdot T \Rightarrow NaryR\ zero \cdot X$ 
| naryRS :  $\forall Ih : \star. \forall n : Nat. NaryR\ n \cdot Ih \rightarrow$ 
       $\forall X : \star. TpEq \cdot X \cdot (T \rightarrow Ih) \Rightarrow NaryR\ (succ\ n) \cdot X$ 
```

3.2 Proof that $NaryR$ is a Functional Relation

We now overview the proof that $NaryR$ is a functional relation, shown partially in Figure 5 and sketched below (the full Cedille proof can be found in the code repository). Though we omit many details of the machine-checked derivation from the code listing, we give proof sketches in natural language to convey the essence of the derivation.

```

naryRResp : ∀ n: Nat. ∀ T1: *. NaryR n ·T1 → ∀ T2: *. Tpeq ·T1 ·T2 ⇒ NaryR n ·T2

naryRUnique : ∀ n: Nat. ∀ T1: *. NaryR n ·T1 → ∀ T2: *. NaryR n ·T2 → Tpeq ·T1 ·T2

naryZEq : Tpeq ·(Nary zero) ·T
naryZ : NaryR zero ·(Nary zero)

narySEq : ∀ n: Nat. NaryR n ·(Nary n) → Tpeq ·(Nary (succ n)) ·(T → Nary n)
naryS : ∀ n: Nat. NaryR n ·(Nary n) → NaryR (succ n) ·(Nary (succ n))

naryREx : Π n: Nat. NaryR n ·(Nary n)
naryREx zero = naryZ
naryREx (succ n) = naryS -n (naryREx n)

```

■ **Figure 5** Respectfulness, uniqueness, and existence [source].

Having made the operating notion of type equality extensional, we are required to prove another property (in addition to uniqueness and existence): it *respects* (or *is congruent with*) extensional type equality.

► **Proposition 1** (Respectfulness (`naryRResp`)). *For all $n : \text{Nat}$ and $T_1, T_2 : *$, if Nary relates n to T_1 and T_1 is equal to T_2 , then Nary relates n to T_2 also.*

Proof idea. By case analysis on the assumed proof $x : \text{NaryR } n \cdot T_1$. In both cases, we have a type X which is equal to T_1 , so use transitivity to conclude X is equal to T_2 . ◀

► **Proposition 2** (Uniqueness (`naryRUnique`)). *For all $n : \text{Nat}$ and $T_1, T_2 : *$, if Nary relates n to T_1 and also to T_2 , then T_1 and T_2 are equal.*

Proof idea. By induction on the assumed proofs $f_1 : \text{NaryR } n \cdot T_1$ and $f_2 : \text{NaryR } n \cdot T_2$. In the case for `naryRZ`, T_1 and T_2 are equal to T and thus to each other. In the case for `naryRS`, T_1 is equal to a type of the form $T \rightarrow \text{Ih}_1$ and T_2 is equal to a type of the form $T \rightarrow \text{Ih}_2$ for some $\text{Ih}_1, \text{Ih}_2 : *$, both of which are assumed to satisfy $\text{Nary } n'$ (where $n = \text{succ } n'$). By the inductive hypothesis, Ih_1 and Ih_2 are equal. Since the type constructor \rightarrow respects type equality in both domain and codomain, we have $T \rightarrow \text{Ih}_1$ is equal to $T \rightarrow \text{Ih}_2$, and thus T_1 is equal to T_2 . ◀

Compared to the first two properties, the proof of existence, `naryEx`, is more involved. It proceeds by induction, using lemmas `naryZ` and `naryS`, which specialize the constructors `naryRZ` and `naryRS` to the corresponding members of the *Nary* family. We only sketch the idea of one of these two lemmas, `naryRS` (the idea for `naryRZ` appeared in Section 3.1).

► **Lemma 3** (`naryS`). *For all $n : \text{Nat}$, if NaryR relates n and $\text{Nary } n$, then it relates $\text{succ } n$ and $\text{Nary } (\text{succ } n)$.*

Proof idea. First, given the assumption that $\text{NaryR } n \cdot (\text{Nary } n)$ holds, we have $\text{NaryR } (\text{succ } n) \cdot (T \rightarrow \text{Nary } n)$ as an instance of the constructor `naryRS`. From this and the proof that NaryR respects type equality, it suffices to show that $\text{Nary } (\text{succ } n)$ and $T \rightarrow \text{Nary } n$ are equal types. We proceed by proving a two-way type inclusion.

■ In the first direction, we assume $f : \text{Nary } (\text{succ } n)$. Since this type is the intersection of the family of types X such that $\text{NaryR } (\text{succ } n) \cdot X$ holds, we conclude by showing that $T \rightarrow \text{Nary } n$ is in this family; this allows us to assign this type to f .

9:8 Simulating Large Eliminations in Cedille

```

naryZC : Nary zero → T
naryZC = tpEq1 -naryZEq

narySC : ∀ n: Nat. Nary (succ n) → (T → Nary n)
narySC -n = tpEq1 -(narySEq -n (naryREx n))

naryZCId : { naryZC = λ x. x }
naryZCId = β

narySCId : ∀ n: Nat. { narySC -n ≈ λ x. x }
narySCId -n = β

```

■ **Figure 6** Computation laws for *Nary* as zero-cost coercions [source].

■ In the second direction, we assume $f : T \rightarrow \text{Nary } n$ and an arbitrary type X such that $\text{Nary } (\text{succ } n) \cdot X$ holds, and must show f can be assigned the type X . We appeal to uniqueness, as NaryR relates $\text{succ } n$ to both X and $T \rightarrow \text{Nary } n$. Since X and $T \rightarrow \text{Nary } n$ are equal, f can be assigned type X . ◀

► **Proposition 4** (Existence (naryREx)). *For all $n : \text{Nat}$, NaryR relates n and $\text{Nary } n$.*

Proof. By induction on n , using lemmas naryZ and naryS . ◀

3.3 Computation Laws as Zero-cost Type Coercions

The proof of existence, naryREx , takes time linear in its argument n to compute a proof of $\text{NaryR } n \cdot (\text{Nary } n)$. Therefore, at first glance it would seem that any type coercions using naryREx could not be constant time. However, thanks to erasure in CDLE this is *not* the situation: eliminators tpEq1 and tpEq2 (Figure 3) take the proof of type equality as an *erased* argument, meaning the runtime complexity of naryREx is irrelevant to the type coercion to which it entitles us!

Figure 6 demonstrates concretely the above discussion. Therein, we define the type coercions naryZC and narySC , corresponding to the two computation laws (left-to-right) for NaryR . In the definition of narySC , note that n is bound as an erased argument, and that our problematic linear-time proof naryREx occurs only as part of the *erased* argument to tpEq1 . Furthermore, we are able to *prove* that these two coercions are equal to the identity function. The proofs, named naryZCId and narySCId in the figure, are given by β in both cases, meaning that this equality holds not just propositionally, but *definitionally* – as we would expect given the erasure rules for tpEq1 .

► **Example 5.** We conclude with an example: applying an n -ary function to n arguments of type T , given as a length-indexed list (Vec). This is shown as appN below.

```

appN : ∀ n: Nat. Nary n → Vec · T n → T
appN -zero    f vnil          = naryZC f
appN -(succ n) f (vcons -n x xs) = appN -n ((narySC -n f) x) xs

```

The definition proceeds by induction on the list of arguments of type $\text{Vec} \cdot T \ n$. In the vcons case, the given natural number is revealed to have the form $\text{succ } n$, so we may coerce the type of $f : \text{Nary } (\text{succ } n)$ to the type $T \rightarrow \text{Nary } n$ to may apply f to the head of the list, then recursively call appN on the tail.

4 Generic Programming Case Studies

In the previous section, we outlined the recipe simulating large eliminations, and in particular we showed explicitly the use of type coercions for the example of applying an n -ary function. For the case studies we consider next, all code listings are presented in a syntax that omits the uses of type coercions to improve readability. In our implementation, we must explicitly use these coercions as well as several substitution lemmas for $TPEq$ over type constructors. As CDLE is a kernel theory (and thus not intended to be ergonomic to program in), the purpose of these examples is to show that this simulation is indeed capable of expressing common generic programming tasks, and we leave the implementation of a high-level surface language for its utilization as future work. We do, however, remark on any new difficulties that are obscured by this presentation (such as Remark 9). Full details of all examples of this section can be found in the repository associated with this paper.

4.1 A Closed Universe of Strictly Positive Datatypes

```

data Descr : *
= idD      : Descr
| constD   : Descr
| pairD    : Descr → Descr → Descr
| sumD     : Π c : C. (I c → Descr) → Descr
| sigD     : Π n : Nat. (Fin n → Descr) → Descr

Decode : * → Descr → *
Decode ·T idD          = T
Decode ·T constD      = Unit
Decode ·T (pairD d1 d2) = Pair ·(Decode ·T d1) ·(Decode ·T d2)
Decode ·T (sumD c f)   = Sigma ·(I c) ·(λ i : I c. Decode ·T (f i))
Decode ·T (sigD n f)   = Sigma ·(Fin n) ·(λ i : Fin n. Decode ·T (f i))

U : Descr → *
U d = μ (λ T : *. Decode ·T d)

inSig : ∀ n : Nat. ∀ cs : Fin n → Descr. Π i : Fin n. U (cs i) → U (sigD n cs)
inSig -n -cs i d = in (i , d)

```

■ **Figure 7** A closed universe of strictly positive types [Descr source] [Decode source].

In the preceding section, we saw an example of arity-generic programming. We consider now a *type-generic* task: proving the *no confusion* property [5] of datatype constructors for a closed universe of strictly positive types. For the datatype universe, the idea (describe in more detail by Dagand and McBride [9]) is to define a type whose elements are interpreted as codes for datatype signatures and combine this with a type-level least fixedpoint operator.

This universe is shown in Figure 7, where $Descr$ is the type of codes for signatures, $Decode$ the large elimination interpreting them, and $C : *$ and $I : C \rightarrow *$ are parameters to the derivation. Signatures comprise the identity functor (idD), a constant functor returning the unitary type $Unit$ ($constD$), a product of signatures ($pairD$), and two forms of sums. The latter of these, $sigD$, takes an argument $n : Nat$ for the number of constructors and a family of n descriptions of the constructor argument types ($Fin n$ is the type of natural numbers less than n). The former, $sumD$, is a more generalized form that takes a code $c : C$

9:10 Simulating Large Eliminations in Cedille

for a constructor argument type, and a mapping of values of type $I\ c$ (where I interprets these codes) to descriptions. Both are interpreted by *Decode* as dependent pairs which pack together an element of the indexing type ($I\ c$ or $Fin\ n$) with the decoding of the description associated with that index.

► **Remark 6.** In order to express a variety of datatypes, our universe is parameterized by codes C and interpretations $I : C \rightarrow \star$ for constructor argument types, such as used in Example 8 below. Unlike much of the literature describing the definition of a closed universe of strictly positive types [6, 9, 8] wherein the host language is a variation of intrinsically typed Martin-Löf type theory, CDLE is *extrinsically typed* – type arguments to constructors can play no role in computation, *even* in the (simulated) computation of other types. This appears to be essential for avoiding paradoxes of the form described by Coquand and Paulin [7], as CDLE is an impredicative theory in which datatype signatures need not be strictly positive.

Finally, the family of datatypes within this universe is given as U , defined using a type-level least fixedpoint operator μ which we discuss in more detail in Section 5. We define a constructor *inSig* for datatypes whose signatures are described by codes of the form $sigD\ n\ cs$ (for $n : Nat$ and $cs : Fin\ n \rightarrow Descr$) using the generic constructor $in : F\ \mu F \rightarrow \mu F$.

► **Example 7 (Natural numbers).** The type of natural numbers can be defined as:

```
unatSig : Descr
unatSig = sigD 2 (fvcons constD (fvcons idD fvnil))
```

```
UNat = U unatSig
```

where *fvcons* and *fvnil* are utilities for expressing functions out of $Fin\ n$ in a list-like notation.

The constructors of $UNat$ are:

```
uzero : UNat
uzero = inSig fin0 unit
```

```
usucc : UNat → UNat
usucc n = inSig fin1 n
```

We do not need the parameters C and I for these definitions.

► **Example 8 (Lists).** Let $T : \star$ be an arbitrary type, and let parameters C and I be resp. $Unit$ and $\lambda\ _ . T$. The type of lists containing elements of type T is defined as:

```
ulistSig : Descr
ulistSig = sigD 2 (fvcons constD (fvcons (sumD unit (\ \_ . idD)) fvnil))
```

```
UList = U ulistSig
```

with constructors defined similarly to those of $UNat$ in the preceding example.

Proving No Confusion

Figure 8 shows the definition of the *no confusion* property, $NoConfusion$, as well as the type of the proof $noConfusion$ which states that the property holds for all equal datatype values. $NoConfusion$ is defined by case analysis over the two datatype values, and additionally abstracts over a test of equality between the constructor labels $i1$ and $i2$. The clause in which they are equal corresponds to the statement of constructor injectivity (the two terms are equal only if equal arguments were given to the constructor); the clause where $i1 \neq i2$ gives the statement of disjointness (datatype expressions cannot be equal and also be in the image of distinct constructors). The proof $noConfusion$ (definition omitted) proceeds by abstracting over the same equality test, and in both cases relies on injectivity of *inSig*.

```

NoConfusion :  $\Pi n: \text{Nat}. \Pi cs: \text{Fin } n \rightarrow \text{Descr}. U (\text{sigD } n \text{ } cs) \rightarrow U (\text{sigD } n \text{ } cs) \rightarrow \star$ 
NoConfusion n cs (in (i1 , d1)) (in (i2 , d2)) | i1 =? i2
NoConfusion n cs (in (i1 , d1)) (in (i1 , d2)) | yes _ = { d1  $\simeq$  d2 }
NoConfusion n cs (in (i1 , d1)) (in (i2 , d2)) | no _ = False

noConfusion :  $\forall n: \text{Nat}. \forall cs: \text{Fin } n \rightarrow \text{Descr}.
  \Pi d1: U (\text{sigD } n \text{ } cs). \Pi d2: U (\text{sigD } n \text{ } cs).
  \{ d1 \simeq d2 \} \rightarrow \text{NoConfusion } d1 \text{ } d2$ 

```

■ **Figure 8** Statement and proof of *no confusion* [source].

```

 $\kappa\text{TpVec } (n : \text{Nat}) = \text{Fin } n \rightarrow \star$ 

TVNil :  $\kappa\text{TpVec } \text{zero}$ 
TVNil _ =  $\forall X: \star. X$ .

TVCons :  $\Pi n: \text{Nat}. \Pi H: \star. \Pi L: \kappa\text{TpVec } n \rightarrow \kappa\text{TpVec } (\text{succ } n)$ 
TVCons n  $\cdot$  H  $\cdot$  L zeroFin = H
TVCons n  $\cdot$  H  $\cdot$  L (succFin i) = L i

TVHead :  $\Pi n: \text{Nat}. \kappa\text{TpVec } (\text{succ } n) \rightarrow \star$ 
TVHead n  $\cdot$  L = L zeroFin

TVTail :  $\Pi n: \text{Nat}. \kappa\text{TpVec } (\text{succ } n) \rightarrow \kappa\text{TpVec } n$ 
TVTail n  $\cdot$  L i = L (succFin i)

TVMap :  $\Pi F: \star \rightarrow \star. \Pi n: \text{Nat}. \kappa\text{TpVec } n \rightarrow \kappa\text{TpVec } n$ 
TVMap  $\cdot$  F n  $\cdot$  L i = F  $\cdot$  (L i)

TVFold :  $\Pi F: \star \rightarrow \star \rightarrow \star. \Pi n: \text{Nat}. \kappa\text{TyVec } (\text{succ } n) \rightarrow \star$ 
TVFold  $\cdot$  F zero  $\cdot$  L = TVHead zero  $\cdot$  L
TVFold  $\cdot$  F (succ n)  $\cdot$  L = F  $\cdot$  (TVHead n  $\cdot$  L)  $\cdot$  (TVFold n  $\cdot$  (TVTail (succ n)  $\cdot$  L))

```

■ **Figure 9** Vectors of types [source].

► **Note.** Though our definition of *NoConfusion* follows that of Dagand and McBride [9], it has a subtle difference: the primitive equality type in CDLE is *untyped*. Specifically, in the case where we have that i_1 and i_2 are equal, we do not need the evidence of this fact to make $\{d_1 \simeq d_2\}$, the type of equalities between $d_1 : U (cs \ i_1)$ and $d_2 : U (cs \ i_2)$, well-formed.

4.2 Arity-generic Map Operation

The last case study we consider is an arity-generic vector operation that generalizes *map*. We summarize the goal (Weirich and Casinghino [34] give a more detailed explanation): define a function which, for all n and families of types $(A_i)_{i \in \{1 \dots n+1\}}$, takes an n -ary function of type $A_1 \rightarrow \dots \rightarrow A_n \rightarrow A_{n+1}$ and n vectors of type $\text{Vec} \cdot A_i \ m$ (for arbitrary m and $i \in \{1, \dots, n\}$), and produces a result vector of type $\text{Vec} \cdot A_{n+1} \ m$. Note that when $n = 1$, this is the usual map operation, and when $n = 2$ it is *zipWith* (when $n = 0$, we have *repeat* : $\Pi m: \text{Nat}. A_1 \rightarrow \text{Vec} \cdot A_1 \ m$).

9:12 Simulating Large Eliminations in Cedille

```

RespTpEq2 :  $\Pi F: \star \rightarrow \star \rightarrow \star. \star$ 
RespTpEq2 ·F =  $\forall A1: \star. \forall A2: \star. \text{TpEq} \cdot A1 \cdot A2 \Rightarrow$ 
                $\forall B1: \star. \forall B2: \star. \text{TpEq} \cdot B1 \cdot B2 \Rightarrow$ 
                $\text{TpEq} \cdot (F \cdot A1 \cdot B1) \cdot (F \cdot A2 \cdot B2)$ 

tvFoldZEq :  $\forall F: \star \rightarrow \star \rightarrow \star. \text{RespTpEq2} \cdot F \Rightarrow$ 
             $\forall X: \star. \text{TpEq} \cdot (\text{TVFold} \cdot F \text{ zero} \cdot (\text{TVCons} \text{ zero} \cdot X \cdot \text{TVNil})) \cdot X$ 

tvFoldSEq :  $\forall F: \star \rightarrow \star \rightarrow \star. \text{RespTpEq2} \cdot F \Rightarrow$ 
             $\forall n: \text{Nat}. \forall X: \star. \forall L: \kappa\text{TyVec} (\text{succ } n).$ 
             $\text{TpEq} \cdot (\text{TVFold} \cdot F (\text{succ } n) \cdot (\text{TVCons} (\text{succ } n) \cdot X \cdot L)) \cdot (F \cdot X \cdot (\text{TVFold} \cdot F \text{ n} \cdot L))$ 

```

■ **Figure 10** Variant computation laws for *TVFold* [source].

4.2.1 Vectors of Types

Our first task is to represent *Nat*-indexed families – i.e., length-indexed lists, or *vectors* – of types. As discussed in Remark 6, it is not possible to define vectors of types which support lookup as a Cedille datatype. We instead use simulated large eliminations to define them directly as lookup functions. This definition, along with some operations, is shown in Figure 9.

The kind of length n vectors of types, $\kappa\text{TpVec } n$, is defined as a function from $\text{Fin } n$ to \star . For the empty type vector TVNil , it does not matter what type we give for the right-hand side of the equation as Fin zero is uninhabited. For TVCons , we use a (non-recursive) large elimination of the given index, returning the head type H if it is zero and performing a lookup in the tail vector L otherwise. The destructors TVHead and TVTail and the mapping function TVMap are defined as expected. The fold operation, TVFold , is given as a large elimination of the *Nat* argument; in the successor case, the recursive call is made on the tail of the given type vector L .

► **Remark 9.** Somewhat hidden by our use of high-level pseudocode is the fact that, since type equality does not admit a general substitution principle, effective use of TVFold requires restricting its first argument to type constructors $F : \star \rightarrow \star \rightarrow \star$ which support substitution with type equality. In particular, if we do not make this assumption, then for types of the form $\text{TVFold} \cdot F (\text{succ } (\text{succ } n)) \cdot L$ we can in general simulate only *one* computation step.

Additionally, under the assumption type constructor F respects type equality in both its type arguments, we are able to give an alternative, more familiar characterization of $\text{TVFold} \cdot F$ by expressing its action over type vectors constructed from TVNil and TVCons . We show this characterization in Figure 10, where $\text{RespTpEq2} \cdot F$ formally expresses that F respects type equality, and tvFoldZEq and tvFoldSEq respectively express the action of TVFold on a singleton list and a list with two or more arguments.

4.2.2 ArrTp and nvecMap

We are now ready to define the arity-generic vector operation nvecMap , shown in Figure 11. We begin with ArrTp , the large elimination that computes the type $A_1 \rightarrow \dots \rightarrow A_n \rightarrow A_{n+1}$ as a fold over a vector of types $L = (A_i)_{i \in \{1 \dots n+1\}}$. The type $\text{Vec} \cdot A_1 \text{ } m \rightarrow \dots \rightarrow \text{Vec} \cdot A_n \text{ } m \rightarrow \text{Vec} \cdot A_{n+1} \text{ } m$ is then constructed simply by composing $\text{ArrTp } n$ with a map over L taking each entry A_i to the type $\text{Vec} \cdot A_i \text{ } m$, shown in ArrTpVec .


```

ArrTp :  $\Pi n: \text{Nat}. \kappa\text{TpVec} (\text{succ } n) \rightarrow \star$ 
ArrTp = TVFold  $\cdot (\lambda X: \star. \lambda Y: \star. X \rightarrow Y)$ 

ArrTpVec m n  $\cdot L$  = ArrTp n  $\cdot (\text{TVMMap} \cdot (\lambda A: \star. \text{Vec} \cdot A \ m) (\text{succ } n) \cdot L)$ 

vrepeat :  $\forall A: \star. \Pi m: \text{Nat}. A \rightarrow \text{Vec} \cdot A \ m$ 
vapp    :  $\forall A: \star. \forall B: \star. \forall m: \text{Nat}. \text{Vec} \cdot (A \rightarrow B) \ m \rightarrow \text{Vec} \cdot A \ m \rightarrow \text{Vec} \cdot B \ m$ 

nvecMap :  $\Pi m: \text{Nat}. \Pi n: \text{Nat}. \forall L: \kappa\text{TpVec} (\text{succ } n). \text{ArrTp } n \cdot L \rightarrow \text{ArrTpVec } m \ n \cdot L$ 
nvecMap m n  $\cdot L$  f = go n  $\cdot L$  (vrepeat m f)
  where
  go :  $\Pi n: \text{Nat}. \forall L: \kappa\text{TpVec} (\text{succ } n) \rightarrow \text{Vec} \cdot (\text{ArrTp } n \cdot L) \rightarrow \text{ArrTpVec } m \ n \cdot L$ 
  go zero  $\cdot L$  fs = fs
  go (succ n)  $\cdot L$  fs =  $\lambda xs. \text{go } n \cdot (\text{TVTail} (\text{succ } n) \cdot L) (\text{vapp } -m \ fs \ xs)$ 

```

■ **Figure 11** Arity-generic map [source].

For *nvecMap*, we use *vrepeat* to create m replicas of the given n -ary function argument f , then invoke the helper function *go* which is defined by recursion over n . In the zero case, fs has type $\text{Vec} \cdot (\text{TVHead } \text{zero} \cdot L) \ m$, which is equal to the expected type (by the computation laws for *ArrTp* and a proof that *Vec* respects type equality). In the successor case, fs is a vector of functions where the type of each element is equal to

$$\text{TVHead} (\text{succ } n) \cdot L \rightarrow \text{ArrTp } n \cdot (\text{TVTail} (\text{succ } n) \cdot L)$$

and the expected type is

$$\text{Vec} \cdot (\text{TVHead} (\text{succ } n) \cdot L) \ m \rightarrow \text{ArrTpVec } m \ n \cdot (\text{TVTail} (\text{succ } n) \cdot L)$$

so we assume such a vector xs , use *vapp* to apply each function of fs point-wise to the elements of xs , then recurse to consume the remaining arguments.

5 Generic Simulation

We now generalize the approach outlined in Section 3 and simulate large eliminations *generically* for datatypes. In the Cedille tool, datatype declarations are elaborated [16] to impredicative encodings provided by the generic framework of Firsov et al. [12]. This framework is based on the categorical semantics of datatypes as initial algebras [15], specifically *Mendler-style* algebras [32], and it supports a broad class of datatypes including those that are nonstrictly positive. To enjoy this same generality and to establish a foundation for surface-language syntax of large eliminations in the Cedille tool, the developments in this section also uses the framework of op. cit. Specifically, we simulate large eliminations for all datatypes of the form $\mu F : \star$, where $F : \star \rightarrow \star$ is a covariant (but otherwise arbitrary) type scheme and μ is the operator for type-level least fixedpoints provided by Firsov et. al. [12].

We first review Mendler-style recursion, and the framework of op. cit. for inductive Mendler-style lambda encodings of datatypes in CDLE. Then, we define the notion of a Mendler-style F -algebra at the level of types, overcoming a technical difficulty for classical F -algebras arising from CDLE's truncated sort hierarchy. Finally, we show that if a type-level Mendler F -algebra A satisfies a certain condition with respect to derived type equality, then A can be used for a simulated large elimination.

5.1 Mendler-style Recursion and Encodings

We briefly review the datatype recursion scheme *à la* Mendler. Originally proposed by Mendler [23] as a method of impredicatively encoding datatypes, Uustalu and Vene have shown that it forms the basis of an alternative categorical semantics of inductive datatypes [32], and the same have advocated for the Mendler style of coding recursion, arguing that it is more idiomatic than the classical formulation of structured recursion schemes [33].

► **Definition 10** (Mendler-style primitive recursion). *Let $F : \star \rightarrow \star$ be a positive type scheme. The datatype with signature F is μF with constructor $in : F \cdot \mu F \rightarrow \mu F$. The Mendler-style primitive recursion scheme for μF is described by the typing and computation law given for rec below:*

$$\frac{\Gamma \vdash T : \star \quad \Gamma \vdash a : \forall R : \star. (R \rightarrow \mu F) \rightarrow (R \rightarrow T) \rightarrow F \cdot R \rightarrow T}{\Gamma \vdash rec \cdot T \ a : \mu F \rightarrow T}$$

$$rec \cdot T \ a \ (in \ d) \rightsquigarrow a \cdot \mu F \ (id \cdot \mu F) \ (rec \cdot T \ a) \ d$$

In Definition 10, the type T (the *carrier*) is the type of results we wish to compute, and the term a (the *action*) gives a single step of a recursive function, and we call the two of them together a Mendler-style F -algebra for recursion. We understand the type argument R of the action as a kind of subtype of the datatype μF – specifically, a subtype containing only predecessors on which we are allowed to make recursive calls.

The first term argument of the action, a function of type $R \rightarrow \mu F$, we can view as the coercion that realizes the subtyping relation; in the computation law, type argument R is instantiated to μF , and the coercion is just $id \cdot \mu F$, the identity. The next argument, a function of type $R \rightarrow T$, is the handle for making recursive calls; in the computation law, it is instantiated to $rec \cdot T \ a$. Finally, the last argument is an “ F -collection” of predecessors of the type R ; in the computation law, it is instantiated to the collection of predecessors $d : F \ \mu F$ of the datatype value $in \ d$.

► **Note.** We can use the fact that the “coercion” function for Mendler recursion is always instantiated to the identity. In such cases, in CDLE it is idiomatic to have, instead of a computationally relevant argument of type $R \rightarrow \mu F$, a computationally *irrelevant* argument of type $Cast \cdot R \cdot \mu F$ (Figure 2). Since a $Cast$ term is a proof of a type inclusion, this makes explicit the previously informal intuition that the quantified type R is a subtype of μF .

Generic framework for Mendler-style datatypes. Figure 12 gives an axiomatic summary of the generic framework of Firsov et al. [12] for deriving efficient Mendler-style lambda encodings of datatypes with induction. In all inference rules save the type formation rule of μ , the datatype signature F is required to be *Monotonic* (that is, positive).

- in is the datatype constructor. For the developments in this section, we find the Mendler-style presentation given in the figure more convenient than the classical type of in .
- out is the datatype destructor, revealing the F -collection of predecessors used to construct the given value.
- $PrfAlg$ is a generalization of Mendler-style algebras to dependent types. Compared to the earlier discussion:
 - the carrier is a predicate $P : \mu F \rightarrow \star$ instead of a type;
 - the coercion function $R \rightarrow \mu F$ from Mendler recursion becomes an erased witness of type $Cast \cdot R \cdot \mu F$;

$$\begin{aligned}
\text{Monotonic} \cdot F &= \forall X:\star. \forall Y:\star. \text{Cast} \cdot X \cdot Y \Rightarrow \text{Cast} \cdot (F \cdot X) \cdot (F \cdot Y) \\
\text{PrfAlg} \cdot F \cdot m \cdot P &= \forall R:\star. \forall c: \text{Cast} \cdot R \cdot \mu F. \\
&\quad (\Pi x:R. P (\text{cast} -c x)) \rightarrow \Pi xs:F \cdot R. P (\text{in} -m -c xs)
\end{aligned}$$

$$\frac{\Gamma \vdash F : \star \rightarrow \star}{\Gamma \vdash \mu F : \star} \quad \frac{\Gamma \vdash F : \star \rightarrow \star \quad \Gamma \vdash m : \text{Monotonic} \cdot F}{\Gamma \vdash \text{in} -m : \forall R:\star. \text{Cast} \cdot R \cdot \mu F \Rightarrow F \cdot R \rightarrow \mu F}$$

$$\frac{\Gamma \vdash F : \star \rightarrow \star \quad \Gamma \vdash m : \text{Monotonic} \cdot F}{\Gamma \vdash \text{out} -m : \mu F \rightarrow F \cdot \mu F}$$

$$\frac{\Gamma \vdash F : \star \rightarrow \star \quad \Gamma \vdash m : \text{Monotonic} \cdot F}{\Gamma \vdash \text{ind} -m : \forall P:\mu F \rightarrow \star. \text{PrfAlg} \cdot F \cdot m \cdot P \rightarrow \Pi x:\mu F. P x}$$

$$\begin{aligned}
|\text{ind} -m \cdot P a (\text{in} -m \cdot R -c xs)| &=_{\beta\eta} |a \cdot R -c (\lambda x. \text{ind} -m \cdot P a (\text{cast} -c x)) xs| \\
|\text{out} -m (\text{in} -m \cdot R -c xs)| &=_{\beta\eta} |xs|
\end{aligned}$$

■ **Figure 12** Axiomatic summary of the generic framework of Firsov et al. [12].

- given a handle for invoking the inductive hypothesis on predecessors of type R and an F -collection of such predecessors, a P -proof F -algebra action must show that P holds for the value constructed from these predecessors using in .
- ind gives the induction principle: to prove a property P for an arbitrary term of type μF , it suffices to give a P -proof F -algebra.

5.2 Mendler-style Type Algebras

Like other (well-founded) recursive definitions, a large elimination can be expressed as a fold of an algebra. In theories with a universe hierarchy, expressing this algebra is no difficult task: the signature F can be universe polymorphic so that its application to either a type or kind is well-formed. This is *not* the case for CDLE, however, as it has a truncated hierarchy of sorts and no sort polymorphism. More specifically, there is no way to express a classical F -algebra on the level of types, e.g., a kind $(F \star) \rightarrow \star$, as it is not possible to define a function on the level of kinds (which F would need to be).

Thankfully, this difficulty *disappears* when the type algebra is expressed in the Mendler style! This is because F does not need to be applied to the kind (\star) of previously computed types, only to the universally quantified type R . Instead, types are computed from predecessors using an assumption of kind $R \rightarrow \star$.

Figure 13 shows the definition κAlgTy of the kind of Mendler-style algebras for primitive recursion having carrier \star (henceforth we will refer to the action of type algebras simply as *algebra*). Just as in the concrete derivation of Section 3, we require that algebras must respect type equality. This condition is codified in the figure as AlgTyResp , which says:

- given two subtypes R_1 and R_2 of μF (which need *not* be equal),
- and two inductive hypotheses Ih_1 and Ih_2 for computing types from values of type R_1 and R_2 , resp.,
- that return equal types on equal terms, then
- we have that the algebra A returns equal types on equal F -collections of predecessors (where the types of predecessors are resp. R_1 and R_2).

9:16 Simulating Large Eliminations in Cedille

$$\kappa\text{AlgTy} = \Pi R: \star. \text{Cast} \cdot R \cdot \mu F \rightarrow (R \rightarrow \star) \rightarrow F \cdot R \rightarrow \star .$$

$$\begin{aligned} \text{AlgTyResp} &: \kappa\text{AlgTy} \rightarrow \star \\ &= \lambda A: \kappa\text{AlgTy}. \\ &\quad \forall R1: \star. \forall R2: \star. \forall c1: \text{Cast} \cdot R1 \cdot \mu F. \forall c2: \text{Cast} \cdot R2 \cdot \mu F. \\ &\quad \forall \text{Ih1}: R1 \rightarrow \star. \forall \text{Ih2}: R2 \rightarrow \star. \\ &\quad (\Pi r1: R1. \Pi r2: R2. \{ r1 \simeq r2 \} \rightarrow \text{TpEq} \cdot (\text{Ih1 } r1) \cdot (\text{Ih2 } r2)) \rightarrow \\ &\quad \Pi xs1: F \cdot R1. \Pi xs2: F \cdot R2. \{ xs1 \simeq xs2 \} \rightarrow \\ &\quad \text{TpEq} \cdot (A \cdot R1 \ c1 \cdot \text{Ih1 } xs1) \cdot (A \cdot R2 \ c2 \cdot \text{Ih2 } xs2) . \end{aligned}$$

■ **Figure 13** Mendler-style type algebras [source].

► **Remark 11.** A careful reader may have noticed that, in Figure 13, we place no constraints on the witnesses $c_1 : \text{Cast} \cdot R_1 \cdot \mu F$ and $c_2 : \text{Cast} \cdot R_2 \cdot \mu F$, even though they both appear in the final equality of AlgTyResp . This is because none are needed: recall from Figure 2 that etaCast tells us *all* witnesses of a type coercion are provably equal to $\lambda x.x$, so in particular c_1 and c_2 are provably equal to each other.

► **Example 12.** Let $F \cdot R = 1 + R$ be the signature of natural numbers with $\text{zeroF} : \forall R: \star. F \cdot R$ and $\text{succF} : \forall R: \star. R \rightarrow F \cdot R$ the signature's injections. For a given property $P : \mu F \rightarrow \star$, we can express as a fold over a Mendler type algebra the property that P holds for a given value *and* all its predecessors, as might be used for a hypothesis for strong induction. That algebra is given below as StrongIndAlg :

$$\begin{aligned} \text{StrongIndAlg} &: \kappa\text{AlgTy} \\ \text{StrongIndAlg} \cdot R \ c \ \text{Ih} \ (\text{zeroF} \cdot R) &= P \ (\text{in} \ -c \ (\text{zeroF} \cdot R)) \\ \text{StrongIndAlg} \cdot R \ c \ \text{Ih} \ (\text{succF} \cdot R \ n) &= \\ &\quad \text{Pair} \cdot (P \ (\text{in} \ -c \ (\text{succF} \cdot R \ n))) \cdot (\text{StrongIndAlg} \cdot R \ c \ \text{Ih} \ n) \end{aligned}$$

Note that unlike previous examples, this algebra is *recursive* rather than being only *iterative*, as the cast c is used (by *in*) to access predecessors at type μF .

Inspecting this definition, we see it indeed satisfies the condition AlgTyResp , with the proof sketch as follows. We assume $R_1, R_2, c_1 : \text{Cast} \cdot R_1 \cdot \mu F, c_2 : \text{Cast} \cdot R_2 \cdot \mu F$, and $xs_1 : F \cdot R_1$ and $xs_2 : F \cdot R_2$ such that $\{xs_1 \simeq xs_2\}$. We may proceed by considering the cases where both are formed by the same injection.

- In the zeroF case, the algebra returns $P \ (\text{in} \ -c_1 \ (\text{zeroF} \cdot R_1))$ and $P \ (\text{in} \ -c_2 \ (\text{zeroF} \cdot R_2))$, which are convertible by erasure (we do not need etaCast for this).
- In the succF case, Pair respects type equality, so it suffices to prove that the types of the components are equal. From the assumption that $\text{succF} \cdot R_1 \ n_1$ is equal to $\text{succF} \cdot R_2 \ n_2$ (for some $n_1 : R_1, n_2 : R_2$), we obtain $\{n_1 \simeq n_2\}$, allowing us to conclude by using term substitution in the first component type and the inductive hypothesis for the second.

► **Remark 13.** We again note that, in the definition of AlgTyResp , the two assumed subtypes R_1 and R_2 need not be equal. As a consequence, in order to satisfy this condition the type produced by the algebra should *not* depend on its type argument R . A high-level surface language implementation for large eliminations in Cedille could require that the bound type variable R only occurs in type arguments of term subexpressions. As definitional equality of types is modulo erasure of typing annotations in term subexpressions, this would ensure that the meaning (extent) of the type does not depend on R .

```

data FoldR :  $\mu F \rightarrow \star \rightarrow \star$ 
= foldRIn
  :  $\forall R: \star. \forall c: \text{Cast } R \cdot \mu F. \forall xs: F \cdot R.$ 
     $\forall Ih: R \rightarrow \star. (\Pi x: R. \text{FoldR } (\text{cast } -c \ x) \cdot (\text{Ih } x)) \rightarrow$ 
     $\forall X: \star. \text{TpEq } X \cdot (A \cdot R \ c \cdot \text{Ih } xs) \Rightarrow \text{FoldR } (\text{in } -c \ xs) \cdot X$ 

Fold :  $\mu F \rightarrow \star$ 
Fold x =  $\forall X: \star. \text{FoldR } x \cdot X \Rightarrow X$  .

foldRResp   :  $\forall x: \mu F. \forall X1: \star. \text{FoldR } x \cdot X1 \rightarrow \forall X2: \star. \text{TpEq } X1 \cdot X2 \Rightarrow \text{FoldR } x \cdot X2$ 
foldRUnique :  $\forall x: \mu F. \forall X1: \star. \text{FoldR } x \cdot X1 \rightarrow \forall X2: \star. \text{FoldR } x \cdot X2 \rightarrow \text{TpEq } X1 \cdot X2$ 
foldREx     :  $\Pi x: \mu F. \text{FoldR } x \cdot (\text{Fold } x)$ 

foldBeta :  $\forall R: \star. \forall c: \text{Cast } R \cdot \mu F. \forall xs: F \cdot R.$ 
            $\text{TpEq } (\text{Fold } (\text{in } -c \ xs)) \cdot (A \cdot R \ c \cdot (\lambda x: R. \text{Fold } (\text{cast } -c \ x)) \ xs)$ 

foldEta :  $\forall H: \mu F \rightarrow \star.$ 
           $\Pi \text{homH}: \forall R: \star. \forall c: \text{Cast } R \cdot \mu F. \Pi xs: F \cdot \mu F.$ 
             $\text{TpEq } (H (\text{in } -c \ xs)) \cdot (A \cdot R \ c \cdot (\lambda x: R. H (\text{cast } -c \ x)) \ xs).$ 
           $\Pi x: \mu F. \text{TpEq } (H \ x) \cdot (\text{Fold } x)$ 

```

■ **Figure 14** Generic large elimination [source].

5.3 Relational Folds of Type Algebras

Figure 14 gives the definition of *FoldR*, a GADT expressing the fold of a type level algebra $A : \kappa \text{AlgTy}$ over μF as a functional relation (A and F are parameters to the definition). It has a single constructor, *foldRIn*, corresponding to the single generic constructor *in* of the datatype, whose type we read as follows:

- given a subtype R of μF and a collection of predecessors $xs : F \cdot R$, and
- a function $Ih : R \rightarrow \star$ that, for every element x in its domain, produces a type related (by *FoldR*) to that element, then
- the datatype value constructed from xs is related to all types that are equal to $A \cdot R \ c \cdot Ih \ xs$.

Just as in Section 3, to show that the inductive relation given by *FoldR* determines a function (from μF to equivalence classes of types), we define a canonical name (*Fold*) for the types determined by the datatype elements and prove that the relation satisfies three properties: it respects type equality, and every datatype element uniquely determines a type. The proofs of respectfulness and existence properties proceed similarly to the concrete proofs given for n -ary functions (see the code repository for full details). We use the condition on type algebras in the proof of uniqueness, so we give a proof sketch below.

► **Proposition 14** (Uniqueness (*foldRUnique*)). *For all $x : \mu F$ and $X_1, X_2 : \star$, if *FoldR* relates x to both X_1 and X_2 , then X_1 and X_2 are equal.*

Proof idea. By induction on the proofs $f_1 : \text{FoldR } x \cdot X_1$ and $f_2 : \text{FoldR } x \cdot X_2$. In the case for *foldRIn* for f_1 , we know that x is of the form $\text{in} \cdot R_1 \cdot c_1 \ xs_1$ for some $R_1 : \star$, $c_1 : \text{Cast } R_1 \cdot \mu F$, and $xs_1 : F \cdot R$. Similarly, from f_2 we know that x is also of the form $\text{in} \cdot R_2 \cdot c_2 \ xs_2$. By injectivity of *in*, we know that $\{xs_1 \simeq xs_2\}$. Call this result (1).

In the case for *foldRIn*, we also are given: from f_1 , a type family $Ih_1 : R_1 \rightarrow \star$ such that *FoldR* relates every $x : R_1$ (R_1 is a subtype of μF) to $Ih_1 \ x$, and a type X_1 extensionally equal to $A \cdot R_1 \ c_1 \cdot Ih_1 \ xs_1$; from f_2 , a type family Ih_2 and type X_2 satisfying similar conditions. By the inductive hypothesis, we can obtain the fact that for all $r_1 : R_1$ and $r_2 : R_2$ such that $\{r_1 \simeq r_2\}$, $Ih_1 \ r_1$ is equal to $Ih_2 \ r_2$. Call this result (2).

We may now use results (1) and (2) to invoke the assumed condition on A that it respects type equality, obtaining a proof that $A \cdot R_1 \cdot c_1 \cdot Ih_1 \cdot xs_1$ is equal to $A \cdot R_2 \cdot c_2 \cdot Ih_2 \cdot xs_2$. From this and some equational reasoning it follows that X_1 is equal to X_2 , concluding the proof. ◀

► **Remark 15.** At present, we are unable to express in a *single definition* type constructor algebras with arbitrarily kinded carriers. Thus, while our derivation is parametric in a datatype signature, it must be repeated once for each type constructor kind. This process is however entirely mechanical, so an implementation of a higher-level surface language for large eliminations in Cedille could elaborate each variant of the derivation as needed, removing the burden of writing boilerplate code.

5.3.1 Characterization

The last two definitions of Figure 14 characterize *Fold* as a recursion scheme. The computation law, given by *foldBeta*, follows a similar pattern as for the Mendler-style recursion shown in Definition 10. *Fold* acts over a datatype value constructed with predecessors $xs : F \cdot R$ by calling the type level algebra A with *Fold* as the handle for recursive calls. Since *in* is a Mendler-style datatype constructor, we instantiate the type argument of A to R so that A may be applied directly to xs . The requirement that A satisfies *AlgTyResp* means that this is equivalent (up to type equality) to instantiating A with μF and applying this to xs after casting xs to the type $F \cdot \mu F$.

In the case studies of Sections 3 and 4, we discussed only the computation laws of our simulated large eliminations. For the generic result, we go further: *Fold* satisfies (up to type equality and function extensionality) the *extensionality law* for Mendler-style recursion, which says that *Fold* is uniquely defined by its action on the values generated by the constructor *in*. This is shown as *foldEta* in the figure, whose type says that any other function $H : \mu F \rightarrow \star$ that satisfies the same computation law as *Fold* is in fact equal to *Fold*.

6 Related Work

CDLE. In an earlier formulation of CDLE [27], Stump proposed a mechanism called *lifting* which allowed simply typed terms to be lifted to the level of types. While adequate for both proving constructor disjointness for natural numbers and enabling some type-generic programming (such as formatted printing in the style of `printf`), its presence significantly complicated the meta-theory of CDLE and its expressive ability was found to be incomplete [28]. Lifting was subsequently removed from the theory, replaced with the simpler δ axiom for proof discrimination.

Marmaduke et al. [22] described a method of encoding datatype signatures that enables constructor subtyping (*à la* Barthe and Frade [3]) with zero-cost type coercions. A key technique for this result was the use of intersection types and equational constraints to simulate (again with type coercions) the computation of types by case analysis on terms – that is, non-recursive large eliminations. Their method of simulation is therefore suitable for expressing type algebras, but not their folds.

System F_C . The intermediate language used by the Haskell compiler GHC, System F_C [29], is an extension of System F with type coercions and equalities. In particular, within System F_C one can express nonparametric type-level functions by adding type equality axioms, such as $f \text{ Int} \sim \text{Bool}$ (where \sim is the type equality operator for F_C). In our approach, clauses of type-level functions are encoded using datatype constructors, and incoherent or partial functions cannot be used because the relation defined by the underlying datatype must be *proven* to be functional.

MLTT and CC. Smith [26] showed that disjointness of datatype constructors was not provable in Martin-Löf type theory without large eliminations by exhibiting a model of types with only two elements – a singleton set and the empty set. In the calculus of constructions, Werner [35] showed that disjointness of constructors would be contradictory by using an erasure procedure to extract System F^ω terms and types, showing that a proof of $1 \neq 0$ in CC would imply a proof of $(\forall X : \star. X \rightarrow X) \rightarrow \forall X : \star. X$ in F^ω . *Proof irrelevance* is central to both results. Since in CDLE proof relevance is axiomatized with δ , this paper can be viewed as a kind of converse to these results: large eliminations enable proof discrimination, and proof discrimination together with extensional type equality enable the simulation of large eliminations.

GADT Semantics. Our simulation of large eliminations rests upon a semantics of GADTs which (intuitively) interprets them as the least set generated by their constructors. However, the semantics of GADTs is a subject which remains under investigation. Johann and Polonsky [19] recently proposed a semantics which makes them functorial, but in which the above-given intuition fails to hold. In subsequent work, Johann et al. [18] explain that GADTs whose semantics are instead based on impredicative encodings (in which case they are not in general functorial) may be equivalently expressed using explicit type equalities. Though they exclude functorial semantics for GADTs in CDLE, the presence of type equalities (both implicit in the semantics and the explicit uses of derived extensional type equality) are essential for defining a relational simulation of large eliminations.

7 Conclusion and Future Work

We have shown that large eliminations may be simulated in CDLE using a derived extensional type equality, zero-cost type coercions, and GADTs to inductively define functional relations. This result overcomes seemingly significant technical obstacles, chiefly CDLE’s lack of primitive inductive types and universe polymorphism, and is made possible by an axiom for proof discrimination. To demonstrate the effectiveness of the simulation, we examine several case studies involving type- and arity-generic programming. Additionally, we have shown that the simulation may be derived generically (that is, parametric in a datatype signature) with Mendler-style type algebras satisfying a certain condition with respect to type equality.

Syntax. In this paper, we have chosen to present code examples using a high-level syntax to improve readability. While the current version of Cedille [10] supports surface language syntax for datatype declarations and recursion, syntax for large eliminations remains future work. Support for this requires addressing (at least) two issues. First, it requires a sound criterion for determining when the type algebra denoted by the surface syntax satisfies the condition *AlgTyResp* (Section 5.2). We conjecture that a simple syntactic occurrence check, along the lines outlined in Remark 13, for erased arguments will suffice. Second, it is desirable that the type coercions that simulate the computation laws of a large elimination be automatically inferred using a subtyping system based on coercions [21, 30].

Semantics. As discussed in Section 2.2.1, the derived form of extensional type equality used in our simulation lacks a substitution principle. However, we claim that such a principle is validated by CDLE’s semantics [28], wherein types are interpreted as sets of ($\beta\eta$ -equivalence classes of) terms of untyped lambda calculus. Under this semantics, a proof of extensional type equality in the syntax implies equality of the semantic objects. We are therefore optimistic

that CDLE may be soundly extended with a kind-indexed family of type constructor equalities with an extensional introduction form and substitution for its elimination form, removing all limitations of the simulation of large eliminations.

References

- 1 Stuart F. Allen, Mark Bickford, Robert L. Constable, Richard Eaton, Christoph Kreitz, Lori Lorigo, and E. Moran. Innovations in computational type theory using NuPRL. *J. Applied Logic*, 4(4):428–469, 2006. doi:10.1016/j.jal.2005.10.005.
- 2 Robert Atkey. Syntax and semantics of quantitative type theory. In Anuj Dawar and Erich Grädel, editors, *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2018, Oxford, UK, July 09-12, 2018*, pages 56–65. ACM, 2018. doi:10.1145/3209108.3209189.
- 3 Gilles Barthe and Maria João Frade. Constructor subtyping. In S. Doaitse Swierstra, editor, *Programming Languages and Systems, 8th European Symposium on Programming, ESOP'99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, 22-28 March, 1999, Proceedings*, volume 1576 of *Lecture Notes in Computer Science*, pages 109–127. Springer, 1999. doi:10.1007/3-540-49099-X_8.
- 4 Corrado Böhm, Mariangiola Dezani-Ciancaglini, P. Peretti, and Simona Ronchi Della Rocca. A discrimination algorithm inside lambda-beta-calculus. *Theor. Comput. Sci.*, 8:265–292, 1979. doi:10.1016/0304-3975(79)90014-8.
- 5 R. M. Burstall and J. A. Goguen. *Algebras, Theories and Freeness: An Introduction for Computer Scientists*, pages 329–349. Springer Netherlands, Dordrecht, 1982. doi:10.1007/978-94-009-7893-5_11.
- 6 James Chapman, Pierre-Évariste Dagand, Conor McBride, and Peter Morris. The gentle art of levitation. In *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 3–14, 2010. doi:10.1145/1863543.1863547.
- 7 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88, International Conference on Computer Logic, Tallinn, USSR, December 1988, Proceedings*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66. Springer, 1988. doi:10.1007/3-540-52335-9_47.
- 8 Pierre-Évariste Dagand. *A cosmology of datatypes: reusability and dependent types*. PhD thesis, University of Strathclyde, Glasgow, UK, 2013. URL: http://oleg.lib.strath.ac.uk/R/?func=dbin-jump-full&object_id=22713.
- 9 Pierre-Évariste Dagand and Conor McBride. Elaborating inductive definitions. *CoRR*, abs/1210.6390, 2012. arXiv:1210.6390.
- 10 Cedille development team. Cedille v1.2.1. <https://github.com/cedille/cedille>.
- 11 Larry Diehl, Denis Firsov, and Aaron Stump. Generic zero-cost reuse for dependent types. *Proc. ACM Program. Lang.*, 2(ICFP):104:1–104:30, July 2018. doi:10.1145/3236799.
- 12 Denis Firsov, Richard Blair, and Aaron Stump. Efficient Mendler-style lambda-encodings in Cedille. In Jeremy Avigad and Assia Mahboubi, editors, *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10895 of *Lecture Notes in Computer Science*, pages 235–252, Cham, 2018. Springer International Publishing. doi:10.1007/978-3-319-94821-8_14.
- 13 Denis Firsov and Aaron Stump. Generic derivation of induction for impredicative encodings in Cedille. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs, CPP 2018*, pages 215–227, New York, NY, USA, 2018. Association for Computing Machinery. doi:10.1145/3167087.

- 14 Herman Geuvers. Induction is not derivable in second order dependent type theory. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001, Krakow, Poland, May 2-5, 2001, Proceedings*, volume 2044 of *Lecture Notes in Computer Science*, pages 166–181, Berlin, Heidelberg, 2001. Springer. doi:10.1007/3-540-45413-6_16.
- 15 Tatsuya Hagino. *A categorical programming language*. PhD thesis, The University of Edinburgh, UK, 1987.
- 16 Christopher Jenkins, Colin McDonald, and Aaron Stump. Elaborating inductive definitions and course-of-values induction in Cedille, 2019. arXiv:1903.08233.
- 17 Christopher Jenkins and Aaron Stump. Monotone recursive types and recursive data representations in Cedille. *Math. Struct. Comput. Sci.*, 31(6):682–745, 2021. doi:10.1017/S0960129521000402.
- 18 Patricia Johann, Enrico Ghiorzi, and Daniel Jeffries. GADTs, functoriality, parametricity: Pick two. *CoRR*, 2021. arXiv:2105.03389, doi:10.4204/EPTCS.357.6.
- 19 Patricia Johann and Andrew Polonsky. Higher-kinded data types: Syntax and semantics. In *34th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2019, Vancouver, BC, Canada, June 24-27, 2019*, pages 1–13. IEEE, 2019. doi:10.1109/LICS.2019.8785657.
- 20 Alexei Kopylov. Dependent intersection: A new way of defining records in type theory. In *Proceedings of 18th IEEE Symposium on Logic in Computer Science (LICS 2003), 22-25 June 2003, Ottawa, Canada, LICS '03*, pages 86–95. IEEE Computer Society, 2003. doi:10.1109/LICS.2003.1210048.
- 21 Zhaohui Luo. Coercive subtyping. *J. Logic and Computation*, 9(1):105–130, 1999. doi:10.1093/logcom/9.1.105.
- 22 Andrew Marmaduke, Christopher Jenkins, and Aaron Stump. Zero-cost constructor subtyping. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages, IFL 2020*, pages 93–103, New York, NY, USA, 2020. Association for Computing Machinery. doi:10.1145/3462172.3462194.
- 23 N. P. Mendler. Recursive types and type constraints in second-order lambda calculus. In *Proceedings of the Symposium on Logic in Computer Science, (LICS '87)*, pages 30–36, Los Alamitos, CA, June 1987. IEEE Computer Society.
- 24 Alexandre Miquel. The implicit calculus of constructions: Extending pure type systems with an intersection type binder and subtyping. In *Proceedings of the 5th International Conference on Typed Lambda Calculi and Applications, TLCA'01*, pages 344–359, Berlin, Heidelberg, 2001. Springer-Verlag. doi:10.1007/3-540-45413-6_27.
- 25 Jan M. Smith. An interpretation of Martin-Löf's type theory in a type-free theory of propositions. *J. Symb. Log.*, 49(3):730–753, 1984. doi:10.2307/2274128.
- 26 Jan M. Smith. The independence of Peano's fourth axiom from Martin-Lof's type theory without universes. *J. Symb. Log.*, 53(3):840–845, 1988. doi:10.2307/2274575.
- 27 Aaron Stump. The calculus of dependent lambda eliminations. *J. Funct. Program.*, 27:e14, 2017. doi:10.1017/S0956796817000053.
- 28 Aaron Stump and Christopher Jenkins. Syntax and semantics of Cedille, 2018. arXiv:1806.04709.
- 29 Martin Sulzmann, Manuel M. T. Chakravarty, Simon L. Peyton Jones, and Kevin Donnelly. System F with type equality coercions. In François Pottier and George C. Necula, editors, *Proceedings of TLDI'07: 2007 ACM SIGPLAN International Workshop on Types in Languages Design and Implementation, Nice, France, January 16, 2007*, pages 53–66. ACM, 2007. doi:10.1145/1190315.1190324.
- 30 Nikhil Swamy, Michael W. Hicks, and Gavin M. Bierman. A theory of typed coercions and its applications. In Graham Hutton and Andrew P. Tolmach, editors, *Proceeding of the 14th ACM SIGPLAN international conference on Functional programming, ICFP 2009, Edinburgh, Scotland, UK, August 31 - September 2, 2009*, pages 329–340. ACM, 2009. doi:10.1145/1596550.1596598.

- 31 The Coq Development Team. *The Coq Reference Manual, version 8.13*, 2021. Available electronically at <https://coq.github.io/doc/v8.13/refman/>.
- 32 Tarmo Uustalu and Varmo Vene. Mendler-style inductive types, categorically. *Nordic Journal of Computing*, 6(3):343–361, September 1999. URL: <http://dl.acm.org/citation.cfm?id=774455.774462>.
- 33 Tarmo Uustalu and Varmo Vene. Coding recursion a la Mendler (extended abstract). In *Proc. of the 2nd Workshop on Generic Programming, WGP 2000, Technical Report UU-CS-2000-19*, pages 69–85. Dept. of Computer Science, Utrecht University, 2000.
- 34 Stephanie Weirich and Chris Casinghino. Generic programming with dependent types. In Jeremy Gibbons, editor, *Generic and Indexed Programming - International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 217–258. Springer, 2010. doi:10.1007/978-3-642-32202-0_5.
- 35 Benjamin Werner. A normalization proof for an impredicative type system with large elimination over integers. In Bengt Nordström, Kent Petersson, and Gordon Plotkin, editors, *Proc. of the 1992 Workshop on Types for Proofs and Programs*, pages 341–357, June 1992.
- 36 Hongwei Xi, Chiyang Chen, and Gang Chen. Guarded recursive datatype constructors. In Alex Aiken and Greg Morrisett, editors, *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisiana, USA, January 15-17, 2003*, pages 224–235. ACM, 2003. doi:10.1145/604131.604150.

Quantitative Polynomial Functors

Georgi Nakov  

Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK

Fredrik Nordvall Forsberg  

Department of Computer and Information Sciences, University of Strathclyde, Glasgow, UK

Abstract

We investigate containers and polynomial functors in Quantitative Type Theory, and give initial algebra semantics of inductive data types in the presence of linearity. We show that reasoning by induction is supported, and equivalent to initiality, also in the linear setting.

2012 ACM Subject Classification Theory of computation \rightarrow Type theory; Theory of computation \rightarrow Linear logic

Keywords and phrases quantitative type theory, polynomial functors, inductive data types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.10

Supplementary Material *Software (Idris 2 Formalisation):*

<https://github.com/g-nakov/quantitative-poly>

archived at `swh:1:dir:ae7679e3704d07ffa7732029c7c9f0f68abf1271`

1 Introduction

Data types are the basic building blocks of modern type theories and programming languages. Having more powerful data types around can increase the proof-theoretic strength of the theory [34], i.e., allow more programs to be written, and can also make existing proofs/programs more convenient to write [13]. Recent advances in type theories such as cubical type theory [11] have also been accompanied by advances in data type theory, such as quotient and higher inductive types [5, 12, 30]. In this paper, we explore what a corresponding notion of (non-higher, so far) inductive types for the also recently introduced type theory Quantitative Type Theory (QTT) [6, 31] might be. QTT combines dependent types and linear types, in the sense of linear logic [20, 37]. By using linearity to track variable (and hence resource) usage of programs, QTT thus promises to enable formal reasoning about both functional and non-functional correctness of programs. A variant of QTT is implemented in the Idris 2 programming language [8], and we hope that our work can be used as a foundational justification for the implementation of data types there. Conversely, we have used Idris 2 to mechanically verify parts of our development.

In a linear world, there are exciting new data types to explore, such as binary trees where only one subtree at each node is present at runtime. Such trees could be used to reduce memory usage and network traffic, or to accurately model the situation at hand. For example, if the tree represents an I/O program in the sense of Hancock and Setzer [22], where subtrees stand for continuation computations, then we would like to ensure that indeed only one continuation is invoked – we do not want the user to simultaneously launch the nukes, and refrain from doing so, by exploring two different subtrees!

While it would be possible to add such data types to QTT by manually writing down formation, introduction, elimination and computation rules for them in an ad-hoc fashion, this is a cumbersome and error-prone process. The goal of this paper is to present a more principled solution. We would like to *derive* the rules for data types from some kind of canonical rules. Our instinct is to turn to the theory of polynomial functors [18, 19], also known as containers [1]. These specify data types as given by shapes and positions, with



© Georgi Nakov and Fredrik Nordvall Forsberg;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 10; pp. 10:1–10:22

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

10:2 Quantitative Polynomial Functors

a data value given by choosing a shape, and then filling every position of the data type with a payload. Pleasingly, containers are closed under many constructions on types such as products and coproducts, and so all strictly positive data types can be reduced to (fixed points of) containers in “traditional” type theory [2, 15, 25].

Hence we present a quantitative version of containers, replacing dependent pair types with dependent tensor types, and function spaces with linear function spaces. Fixed points of such containers now more or less immediately admit formation and introduction rules for their corresponding data types, but what about the elimination and computation rules? Our main technical contribution is to show that the problem of providing an elimination rule for fixed points of such containers can be reduced to the problem of proving initiality in a category of algebras. For traditional data types, this is well known [23, 7], and our proof is a linear refinement of the traditional proof – pairs with “unused” first components play a key role in the construction.

Is that the end of the story? Unfortunately not! Ordinary containers are useful as a foundation for data types in traditional type theory exactly since all strictly positive data types can be reduced to them. However when we try to replicate this reduction for quantitative data types, we hit a snag: in the quantitative setting, it is no longer the case that every strictly positive type is isomorphic to a quantitative container, because of the splitting up of connectives into additive and multiplicative variants. Quantitative containers still serve as an instructive special case of quantitative data types, but they do not cover every interesting example. Instead, we define quantitative polynomial functors as inductively generated from a grammar containing constants, the identity, and sums and products. We can show that also for this class of functors, elimination rules are supported if and only if the algebra is initial. The proof is similar to the proof for quantitative containers, but now further complicated by having to also do induction on the generation according to the polynomial functors grammar.

Our second major contribution is to show that finitary polynomial functors indeed have initial algebras in Atkey’s linear realisability model of QTT [6, § 4]. This is a model where types are interpreted as *assemblies*, i.e., sets whose elements are assigned realisers from a combinatory algebra of (untyped) linear programs. We construct the initial algebras in two stages: first we consider the initial algebra of the polynomial functor in the category *Set*, which is known to exist. We then use the initiality of the algebra in *Set* to both define the realisability relation, and to prove that the algebra map and mediating morphism into other algebras are realised – this constructs an initial algebra also in the category of assemblies. Putting our contributions together, we have thus defined a syntactic class of polynomial data types, and shown that QTT extended with rules for them, including dependent elimination rules, can be soundly interpreted in Atkey’s realisability model.

Structure of the paper

In Section 2, we recall the syntax and semantics of QTT, including a sketch of a realisability model. In Section 3, we review the traditional theory of initial algebras and containers. We then move from containers to quantitative containers in Section 4, and prove that initiality is equivalent to dependent elimination for them. In Section 5, we generalise quantitative containers to inductively generated quantitative polynomial functors, and show that also for them, initiality is equivalent to dependent elimination. We finally construct initial algebras of finitary quantitative polynomial functors in the aforementioned realisability model.

Partial Idris 2 formalisation

We have formalised the basic definitions and results from Section 4 and Section 5 in Idris 2, most notably including a verification of Theorem 16 and the induction-from-initiality direction of Theorem 26. We make use of Idris 2’s implementation of linearity to faithfully model quantitative containers and quantitative polynomial functors, and to verify that the morphisms we construct, such as dist_F from Lemma 24, really are linear. It was helpful to have Idris 2 guide us by the quantity information *during* the construction of the proofs – for example, feedback from Idris 2 made us realise the correct definition of lifting of constant functors in Definition 20.

The code can be found at <https://github.com/g-nakov/quantitative-poly>.

2 Quantitative Type Theory

In this section, we give a brief introduction to the syntax and semantics of QTT, and present the typing rules for the type formers we will use. For a detailed presentation, see Atkey [6]. In contrast to dependent linear/non-linear type theories [9, 29], QTT maintains a single context in which variables can both contribute to type formation, and be marked as a linear resource. This is in line with recent work by Fu, Kishida and Selinger [17], and Abel and Bernardy [3]. See also Choudhury, Eades III, Eisenberg and Weirich [10], and Orchard, Liepelt and Eades III [32] for similar approaches.

2.1 Syntax of Quantitative Type Theory

The main difference between quantitative and ordinary type theory is that variables in quantitative type theory contexts are annotated with resources, intuitively governing how many times they can be used. These resource annotations are drawn from a semiring, satisfying some additional axioms that are needed for the typing rules to make sense.

► **Definition 1.** A resource semiring is a structure $R = \langle R, +, \cdot, 0, 1 \rangle$, such that $\langle R, +, 0 \rangle$ is a monoid, $\langle R, \cdot, 1 \rangle$ is a commutative monoid, with \cdot distributing over $+$, such that for every $\rho, \pi \in R$, $\rho \cdot \pi = 0$ if and only if $\rho = 0$ or $\pi = 0$, and such that $\rho + \pi = 0$ implies $\rho = \pi = 0$.

Examples of resource semirings are given by the natural numbers with ordinary addition and multiplication, and the “zero-one-many” resource semiring $\langle \{0, 1, \omega^<\}, +, \cdot, 0, 1 \rangle$ with $x + \omega^< = \omega^< + x = \omega^<$, $1 + 1 = \omega^<$, and $\omega^< \cdot \omega^< = \omega^<$.

Fixing a semiring R , a QTT term judgement has the following form:

$$x_1 \overset{\pi_1}{:} S_1, x_2 \overset{\pi_2}{:} S_2, \dots, x_n \overset{\pi_n}{:} S_n \vdash M \overset{\sigma}{:} T$$

with $\pi_i \in R$ and σ restricted to either $\sigma = 0_R$ or $\sigma = 1_R$. The list of variables x_i on the left of the turnstile forms the context of the judgement. The resource semiring operations can be lifted pointwise to contexts:

$$\begin{aligned} \pi(\Gamma, x \overset{\rho}{:} S) &= \pi\Gamma, x \overset{\pi\rho}{:} S \\ (\Gamma_1, x \overset{\rho_1}{:} S) + (\Gamma_2, x \overset{\rho_2}{:} S) &= (\Gamma_1 + \Gamma_2), x \overset{\rho_1 + \rho_2}{:} S, \text{ if } 0\Gamma_1 = 0\Gamma_2. \end{aligned}$$

In typical dependent type theory fashion, the variables x_i are available for forming both types and terms. Each variable x_i is annotated with a resource π_i denoting how many times x_i must be used *computationally* in the term M on the right hand side of the turnstile. In contrast, the calculus is set up so that types are always formed in contexts of the form 0Γ –

10:4 Quantitative Polynomial Functors

$$\begin{array}{c}
\frac{}{0\Gamma \vdash \mathbf{I} : \mathbf{Type}} \quad \frac{}{0\Gamma \vdash \top : \mathbf{Type}} \quad \frac{}{0\Gamma \vdash \mathbf{0} : \mathbf{Type}} \\
\frac{}{0\Gamma \vdash \star : \mathbf{I}} \quad \frac{0\Gamma_1, x : \mathbf{I} \vdash S : \mathbf{Type} \quad \Gamma_1 \vdash m : \mathbf{I} \quad \Gamma_2 \vdash n : S[\star/x]}{\Gamma_1 + \Gamma_2 \vdash \text{let } \star = m \text{ in } n : S[m/x]} \quad \frac{}{\Gamma \vdash \star : \top} \quad \frac{\Gamma \vdash m : \mathbf{0}}{\Gamma \vdash \text{abort } m : B}
\end{array}$$

■ **Figure 1** Type formation and rules for basic types.

type formation consume no resources. The distinction between computational and “vacuous” usage is marked explicitly by the annotation σ on the term M in the conclusion, which can only be 0 or 1. Such a restriction is needed to retain admissibility of substitution and effectively splits the theory in two fragments. Terms in the 0 fragment bear no computational content, consume no resources, and are available at any point in a derivation, while the inhabitants of the 1 fragment are computationally relevant, but need sufficient resources to be constructed. The following meta-result, which states that “zero needs nothing”, is useful for constructing derivations in the 0 fragment, basically without worrying about resource usage:

► **Lemma 2** (Atkey [6]). *If $\Gamma \vdash M : S$, then $\Gamma = 0\Gamma$.*

From now on, we follow the following conventions: Types are formed in the 0 fragment, but we may omit the 0 annotation, that is, we may write $0\Gamma \vdash S : \mathbf{Type}$ for $0\Gamma \vdash S : \mathbf{Type}$. Any other judgement takes place in the 1 fragment, unless explicitly annotated otherwise, and we suppress the annotation on the conclusion – that is, $\Gamma \vdash M : S$ should be read as $\Gamma \vdash M : S$. Scaling the context by 0 yields the corresponding rule in the 0 fragment. We further assume that contexts are well formed; for example, we only consider the context $\Gamma_1 + \Gamma_2$ if $0\Gamma_1 = 0\Gamma_2$.

As an example, consider the type family $\text{Fin} : \mathbb{N} \rightarrow \mathbf{Type}$ of finite types. It has formation rule

$$\frac{0\Gamma \vdash n : \mathbb{N}}{0\Gamma \vdash \text{Fin}(n) : \mathbf{Type}}$$

where we ask for 0 copies of $n : \mathbb{N}$, since this n occurs in a type. The introduction rules are

$$\frac{0\Gamma \vdash n : \mathbb{N}}{0\Gamma \vdash \text{zero} : \text{Fin}(n+1)} \quad \frac{0\Gamma \vdash n : \mathbb{N} \quad \Gamma \vdash m : \text{Fin}(n)}{\Gamma \vdash \text{suc}(m) : \text{Fin}(n+1)}$$

where again $n : \mathbb{N}$ does not require any resources, since it occurs only in a type. We omit the elimination rule here, since we will not make use of it further. Rules for other basic types can be found in Figure 1; the type \mathbf{I} was considered by Atkey [6, §2.1.3], whereas \top and $\mathbf{0}$ are first described for QTT here. The monoidal unit type \mathbf{I} contains no information, and hence it is always possible to construct an element $\star : \mathbf{I}$ at no cost, or use the elimination rule to discard such an element. The terminal type \top superficially looks very similar to the monoidal unit \mathbf{I} , but note that it is possible to construct $\star : \top$ (we reuse the same syntax \star for the term as for \mathbf{I}) even in non-zero contexts, which makes it terminal. There is no elimination rule. Dually the empty type $\mathbf{0}$ allows elimination into any other type.

Rules for type formers are given in Figure 2; $(x : S) \otimes T$ and $(x : S) \rightarrow T$ were considered by Atkey, and $(x : S) \& T$ was introduced by Svoboda [35], whereas $S \oplus T$ is new to QTT, as far as we know. Extra care should be taken in the rules for types with binders, as the bound

$$\begin{array}{c}
\frac{0\Gamma \vdash S : \mathbf{Type} \quad 0\Gamma, x^0 : S \vdash T : \mathbf{Type} \quad \frac{\Gamma, x^{\rho} : S \vdash t : T}{\Gamma \vdash \lambda x. t : (x^{\rho} S) \rightarrow T}}{0\Gamma \vdash (x^{\rho} S) \rightarrow T : \mathbf{Type}} \quad \frac{\Gamma_1 \vdash f : (x^{\rho} S) \rightarrow T \quad \Gamma_2 \vdash s : S}{\Gamma_1 + \rho\Gamma_2 \vdash f(s) : S[s/x]} \\
\\
\frac{0\Gamma \vdash S : \mathbf{Type} \quad 0\Gamma, x^0 : S \vdash T : \mathbf{Type} \quad \frac{\Gamma_1 \vdash s : S \quad \Gamma_2 \vdash t : T[s/x]}{\rho\Gamma_1 + \Gamma_2 \vdash (s, t) : (x^{\rho} S) \otimes T}}{0\Gamma \vdash (x^{\rho} S) \otimes T : \mathbf{Type}} \quad \frac{0\Gamma_1, z^0 : (x^{\rho} S) \otimes T \vdash U : \mathbf{Type} \quad \Gamma_1 \vdash p : (x^{\rho} S) \otimes T \quad \Gamma_2, x^{\rho} : S, y^1 : T \vdash u : U[(x, y)/z]}{\Gamma_1 + \Gamma_2 \vdash \text{let } (x, y) = p \text{ in } u : U[p/z]} \\
\\
\frac{0\Gamma \vdash S : \mathbf{Type} \quad 0\Gamma, x^0 : S \vdash T : \mathbf{Type} \quad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash (s, t) : (x : S) \& T}}{0\Gamma \vdash (x : S) \& T : \mathbf{Type}} \quad \frac{\Gamma \vdash p : (x : S) \& T}{\Gamma \vdash \text{proj}_1(p) : S} \quad \frac{\Gamma \vdash p : (x : S) \& T}{\Gamma \vdash \text{proj}_2(p) : T[\text{proj}_1(p)/x]} \\
\\
\frac{0\Gamma \vdash S : \mathbf{Type} \quad 0\Gamma \vdash T : \mathbf{Type} \quad \frac{\Gamma \vdash s : S}{\Gamma \vdash \text{inl } s : S \oplus T} \quad \frac{\Gamma \vdash t : T}{\Gamma \vdash \text{inr } t : S \oplus T}}{0\Gamma \vdash S \oplus T : \mathbf{Type}} \quad \frac{0\Gamma_1, z^0 : S \oplus T \vdash U : \mathbf{Type} \quad \Gamma_1 \vdash e : S \oplus T \quad \Gamma_2, x^1 : S \vdash u : U[\text{inl } x/z] \quad \Gamma_2, y^1 : T \vdash v : U[\text{inr } y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{case } e \text{ of } \text{inl } x \Rightarrow u \mid \text{inr } y \Rightarrow v : U[e/z]} \\
\\
\frac{0\Gamma \vdash S : \mathbf{Type} \quad 0\Gamma \vdash s^0 : S \quad 0\Gamma \vdash t^0 : S}{0\Gamma \vdash \text{Id}_S(s, t) : \mathbf{Type}} \quad \frac{0\Gamma \vdash s = t : S}{0\Gamma \vdash \text{refl} : \text{Id}_S(s, t)} \quad \frac{0\Gamma \vdash p : \text{Id}_S(s, t)}{0\Gamma \vdash s = t : S}
\end{array}$$

■ **Figure 2** Typing rules for type formers.

variable could potentially represent a computational resource – for example, the dependent function type $(x^{\rho} S) \rightarrow T$ records how many copies ρ of its argument are needed. We write $S \xrightarrow{\rho} T$ for $(x^{\rho} S) \rightarrow T$ if x does not occur in T . The dependent tensor type $(x^{\rho} S) \otimes T$ is the multiplicative linear version of the dependent pair type; it contains pairs (s, t) with ρ copies of s and one copy of t , which must both be consumed fully in the elimination rule. We pay special attention to that type as a key to a simpler presentation of the remaining rules. Using the monoidal unit \mathbf{I} , we can form an exponential type $!_{\rho}S = (x^{\rho} S) \otimes \mathbf{I}$ and “disguise” resource usage information within the types themselves. Thus, we avoid proliferating the typing rules with abundant annotations while still retaining the full expressiveness of QTT. For example, no modifications in the rules are needed to allow for arbitrarily many copies of the second component of the tensor tuple – working with the type $(x^{\rho} S) \otimes (!_{\pi}T)$ already achieves that.

In contrast to the dependent tensor type, the dependent additive conjunction $(x : S) \& T$ (pronounced “with”) contains pairs (s, t) where the consumer can choose to either use $s : S$ or $t : T[s/x]$. We write $S \otimes T$ for $(x^1 : S) \otimes T$ and $S \& T$ for $(x : S) \& T$ respectively, if x does not occur in T – these types correspond to the usual non-dependent linear logic connectives. Finally $S \oplus T$ is the disjoint union of S and T ; note that both branches share the same context Γ_2 in its elimination rule. Combining with exponential types, we can recover the

$$\begin{array}{c}
\frac{\Gamma \vdash n : S[\star/x]}{\Gamma \vdash \text{let } \star = \star \text{ in } n \equiv n : S[\star/x]} \quad \frac{\Gamma_1, x \overset{\rho}{\vdash} S \vdash t : T \quad \Gamma_2 \vdash s : S}{\Gamma_1 + \rho\Gamma_2 \vdash (\lambda x. t) s \equiv t[s/x] : T[s/x]} \\
\frac{\Gamma_1 \vdash s : S \quad \Gamma_2 \vdash t : T[s/x] \quad \Gamma_3, x \overset{\rho}{\vdash} S, y \overset{1}{\vdash} T \vdash u : U[(x, y)/z]}{\rho\Gamma_1 + \Gamma_2 + \Gamma_3 \vdash \text{let } (x, y) = (s, t) \text{ in } u \equiv u[s/x, t/y] : U[(s, t)/z]} \\
\frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \text{proj}_1(s, t) \equiv s : S} \quad \frac{\Gamma \vdash s : S \quad \Gamma \vdash t : T[s/x]}{\Gamma \vdash \text{proj}_2(s, t) \equiv t : T[s/x]} \\
\frac{\Gamma_1 \vdash s : S \quad \Gamma_2, x \overset{1}{\vdash} S \vdash u : U[\text{inl } x/z] \quad \Gamma_2, y \overset{1}{\vdash} T \vdash v : U[\text{inr } y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{case inl } s \text{ of inl } x \Rightarrow u \mid \text{inr } y \Rightarrow v \equiv u[s/x] : U[\text{inl } s/z]} \\
\frac{\Gamma_1 \vdash t : T \quad \Gamma_2, x \overset{1}{\vdash} S \vdash u : U[\text{inl } x/z] \quad \Gamma_2, y \overset{1}{\vdash} T \vdash v : U[\text{inr } y/z]}{\Gamma_1 + \Gamma_2 \vdash \text{case inr } t \text{ of inl } x \Rightarrow u \mid \text{inr } y \Rightarrow v \equiv v[t/y] : U[\text{inr } t/z]} \\
\\
\frac{\Gamma \vdash t : \top}{\Gamma \vdash t \equiv \star : \top} \quad \frac{\Gamma \vdash f : (x \overset{\rho}{\vdash} S) \rightarrow T}{\Gamma \vdash f \equiv \lambda x. f(x) : (x \overset{\rho}{\vdash} S) \rightarrow T}
\end{array}$$

■ **Figure 3** β - and η -rules.

more general version $(!_\rho S) \oplus (!_\pi T)$. We can define the type of Booleans by $\mathbf{2} = \mathbf{I} \oplus \mathbf{I}$. In particular, this implies that the constructors $\text{false} = \text{inl } \star$ and $\text{true} = \text{inr } \star$ can be introduced using zero resources. To facilitate formal reasoning within QTT, we also add extensional identity types. Because of equality reflection, a term p of type $\text{Id}_S(s, t)$ does not use any computational resources, and thus can be introduced even if the resources designated by the terms s and t have been exhausted. We chose to an extensional identity type since they are easier to work with, and easy to model in realisability models – in particular, with an extensional identity type we do not need to worry about if the rules for the identity type should be formulated multiplicatively, additively, or both. In principle one could also try to develop QTT with an intensional flavour – linearity and identity types appear to be orthogonal features in the type system.

Eliminators come with their expected β - and η -rules familiar from ordinary type theory, listed in Figure 3. Note that we have only included η -rules for \top and dependent function types – all other type formers have eliminators that can be used to derive the expected η -rules using the extensional identity type, but \top does not have an eliminator, and the η -rule for dependent function types is needed to derive function extensionality from the extensional identity type.

These rules give types a rich algebraic structure. An isomorphism of types $A \cong B$ is given by two linear functions $\vdash f : A \xrightarrow{1} B$ and $\vdash g : B \xrightarrow{1} A$ such that $x \overset{0}{\vdash} A \vdash g(f(x)) = x : A$ and $y \overset{0}{\vdash} B \vdash f(g(y)) = y : B$ – using equality reflection, these equalities can also be proven using the extensional identity type. For example, the usual currying-uncurrying isomorphism, relating dependent functions and dependent pairs, becomes $(z \overset{1}{\vdash} ((x \overset{\rho}{\vdash} A) \otimes B) \rightarrow C) \cong (x \overset{\rho}{\vdash} A) \rightarrow (y \overset{1}{\vdash} B) \rightarrow C[(x, y)/z]$ in the QTT setting. We will make use of the following isomorphisms.

► **Lemma 3.** *For any type A , we have:*

$$\mathbf{I} \otimes A \cong A \quad \mathbf{0} \xrightarrow{1} A \cong \top \quad \mathbf{I} \xrightarrow{1} A \cong A \quad \mathbf{2} \xrightarrow{1} A \cong A \& A$$

In contrast, for example $\top \otimes A \not\cong A$ and $(A \xrightarrow{1} \mathbf{I}) \not\cong \mathbf{I}$, intuitively because there is no way to consume an element of \top , or to produce a function $A \xrightarrow{1} \mathbf{I}$ in general – this would allow discarding elements of A . Formally, one can show that such isomorphisms do not exist by exhibiting concrete models of QTT where they do not hold.

2.2 Semantics of Quantitative Type Theory

Categories with Families [14] (CwFs) form a sound and complete semantics for dependent type theories, and are given by a category \mathcal{C} (modelling contexts and substitutions), together with a functor $(\text{Ty}, \text{Tm}) : \mathcal{C} \rightarrow \text{Fam}(\text{Set})$ into the category of families of sets (modelling types and terms in a given context), and a context comprehension operation $-._- : (\Gamma : \mathcal{C}) \rightarrow \text{Ty}(\Gamma) \rightarrow \mathcal{C}$ (modelling context extension), with a universal property. Atkey [6] presents a refined notion of *quantitative* CwF to account for QTT’s usage information in contexts, terms and types. Intuitively, a quantitative CwF for a fixed resource semiring R is given by *two* categories \mathcal{L} and \mathcal{C} , used to model resourced contexts and contexts in the 0 fragment, respectively. To any resourced context corresponds a plain one, obtained by forgetting the resource annotations, and so there should be a faithful functor $U : \mathcal{L} \rightarrow \mathcal{C}$. Since the 0 fragment allows unrestricted usage, \mathcal{C} should be a CwF on its own. Types are always formed in the 0 fragment, so there is no need to have a separate notion of “resourced types”, but \mathcal{L} needs to support both resourced context extension and resourced terms, both suitably displayed by U over their unresourced counterparts in \mathcal{C} . Finally, we need to ask for functors $\rho(-) : \mathcal{L} \rightarrow \mathcal{L}$ for each $\rho \in R$, and $(+) : \mathcal{L} \times_{\mathcal{C}} \mathcal{L} \rightarrow \mathcal{L}$ (where $\mathcal{L} \times_{\mathcal{C}} \mathcal{L}$ is the pullback of U along itself), modelling scaling and context addition, respectively.

Since quantitative CwFs are refinements of CwFs, we can construct a “trivially quantitative” CwF for each ordinary CwF \mathcal{C} by putting $\mathcal{L} = \mathcal{C}$ with $U = \text{Id}$, and $\text{RTm} = \text{Tm}$. We can also build models that actually demonstrate the distinction between computational and non-computational use of data. Atkey [6] constructs a model based on linear realisability [24], which we now describe, since we will make use of it in what follows. Unresourced terms will be modelled by set-theoretic functions, while their resourced counterparts will be supplemented with computable, linear realisers, drawn from an R -linear combinatory algebra.

► **Definition 4.** *An R -linear combinatory algebra (R -LCA) consists of a set \mathcal{A} , a binary operation $(\cdot) : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$, a family of unary operations $!_{\rho} : \mathcal{A} \rightarrow \mathcal{A}$ for $\rho \in R$, and combinators $B, C, I, K, D, W_{\pi\rho}, \delta_{\pi\rho}, F_{\rho} \in \mathcal{A}$ for every $\pi, \rho \in R$, satisfying the following equations:*

- $B \cdot x \cdot y \cdot z = x \cdot (y \cdot z)$
- $C \cdot x \cdot y \cdot z = x \cdot z \cdot y$
- $I \cdot x = x$
- $K \cdot x \cdot !_0 y = x$
- $D \cdot !_1 x = x$
- $W_{\pi\rho} \cdot x \cdot !_{\pi+\rho} y = x \cdot !_{\pi} y \cdot !_{\rho} y$
- $\delta_{\pi\rho} \cdot !_{\pi\rho} x = !_{\pi} !_{\rho} x$
- $F_{\rho} \cdot !_{\rho} x \cdot !_{\rho} y = !_{\rho} (x \cdot y)$

An R -LCA \mathcal{A} supports booleans if there exists elements $T, F \in \mathcal{A}$ and a function $\text{case} : \mathcal{A} \times \mathcal{A} \rightarrow \mathcal{A}$, such that for all $p, q \in \mathcal{A}$, $\text{case}(p, q) \cdot T = p$ and $\text{case}(p, q) \cdot F = q$.

10:8 Quantitative Polynomial Functors

The operation \cdot is meant to represent function application; the combinators B, C, I are used to implement composition, exchange and identity, $!_\rho$ represents “making ρ copies”, with $K, D, W_{\pi\rho}, \delta_{\pi\rho}$, and F_ρ implementing structural rules and reshuffling of resources – these witness that $!_- : R \times \mathcal{A} \rightarrow \mathcal{A}$ is an action. See Abramsky, Haghverdi and Scott [4], and Hoshino [24] for more details. Importantly, R -linear combinatory algebras are combinatorially complete: given an expression M built from applications, elements of \mathcal{A} , and variables, with exactly one occurrence of the variable x , there exists an expression $\lambda^*x.M$, not containing x , such that $(\lambda^*x.M) \cdot N = M[N/x]$. We define the tupling of elements $a_1, \dots, a_n \in \mathcal{A}$ using combinatory completeness and the standard Church encoding $[x_1, \dots, x_n] := \lambda^*q.q \cdot x_1 \dots x_n$.

► **Example 5.** Fix R as the zero-one-many semiring $\{0, 1, \omega^<\}$, and let $\langle -, - \rangle : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ be a bijection, and $[-] : \text{List } \mathbb{N} \rightarrow \mathbb{N}$ an encoding of finite lists of natural numbers as natural numbers, in such a way that there is a “list membership” predicate $a \in b$ and a list concatenation $a ++ b$ such that $a_i \in [a_1, \dots, a_n]$, and $[a_1, \dots, a_n] ++ [a_{n+1}, \dots, a_m] = [a_1, \dots, a_m]$ (with no requirements on $a \in bs$ and $as ++ bs$ when as and bs are not in the image of $[-]$). Following Atkey [6, Examples 4.3, 4.6 and 4.7], who in turn followed Hoshino [24, §5.3], we can endow the power set $\mathcal{P}(\mathbb{N})$ of \mathbb{N} with the structure of an R -LCA by defining the application $\alpha \cdot \beta = \{n \mid \langle m, n \rangle \in \alpha, m \in \beta\}$. We write $[a_1, \dots, a_\rho]$ for a list of length ρ , i.e. $[a_1, \dots, a_0] = []$, $[a_1, \dots, a_1] = [a_1]$ and $[a_1, \dots, a_{\omega^<}]$ means $[a_1, \dots, a_n]$ for some $n \in \mathbb{N}$, and we define $!_\rho : \mathcal{P}(\mathbb{N}) \rightarrow \mathcal{P}(\mathbb{N})$ by $!_\rho \alpha = \{[a_1, \dots, a_\rho] \mid a_i \in \alpha\}$. The combinators are given by:

$$\begin{aligned} B &= \{\langle \langle m, k \rangle, \langle \langle n, m \rangle, \langle n, k \rangle \rangle \rangle \mid n, m, k \in \mathbb{N}\} \\ C &= \{\langle \langle m, \langle n, k \rangle \rangle, \langle n, \langle m, k \rangle \rangle \rangle \mid n, m, k \in \mathbb{N}\} \\ I &= \{\langle n, n \rangle \mid n \in \mathbb{N}\} \\ K &= \{\langle n, \langle [], n \rangle \rangle \mid n \in \mathbb{N}\} \\ D &= \{\langle [n], n \rangle \mid n \in \mathbb{N}\} \\ W_{\pi\rho} &= \{\langle \langle a_1, \langle a_2, b \rangle \rangle, \langle a_1 ++ a_2, b \rangle \rangle \mid a_1 = [a_{1,1}, \dots, a_{1,\pi}], a_2 = [a_{2,1}, \dots, a_{2,\rho}], a_{i,j}, b \in \mathbb{N}\} \\ \delta_{\pi\rho} &= \{\langle a_1 ++ \dots ++ a_\pi, [a_1, \dots, a_n] \rangle \mid a_i = [a_{i,1}, \dots, a_{i,\rho}], a_{i,j} \in \mathbb{N}\} \\ F_\rho &= \{\langle f, \langle a, [b_1, \dots, b_\rho] \rangle \rangle \mid \forall b_i. \exists c. c \in a \wedge \langle c, b_i \rangle \in f, a, f \in \mathbb{N}\}. \end{aligned}$$

This R -LCA also supports Booleans by defining $T = \{1\}$, $F = \{0\}$.

As a final prerequisite towards constructing a realisability quantitative CwF, we recall the definition of the category of assemblies [24, §2.3].

► **Definition 6.** Let \mathcal{A} be an R -LCA. An assembly is a pair $\Gamma = (|\Gamma|, \vDash_\Gamma)$, where $|\Gamma|$ is a set, and \vDash_Γ is a relation $\vDash_\Gamma \subseteq \mathcal{A} \times |\Gamma|$. A morphism of assemblies from Γ to Δ is a function $f : |\Gamma| \rightarrow |\Delta|$, which is realisable – there exists $a_f \in \mathcal{A}$, such that $a \vDash_\Gamma \gamma$ implies $a_f \cdot a \vDash_\Delta f(\gamma)$.

We can think of $|\Gamma|$ as a set of extensional meanings, and $a \vDash_\Gamma x$ as saying that the “program” $a \in \mathcal{A}$ realises the meaning of x . Usually, it is required that every $\gamma \in \Gamma$ has a realiser, but Atkey (personal communication) found the need to drop this condition due to technical reasons in the interpretation of dependent function types. This modified notion of assemblies and realisable functions still forms a category, using the B and I combinators to realise composition and identities. We now have all the pieces needed to describe Atkey’s linear realisability model of QTT.

► **Proposition 7.** *Let \mathcal{A} be the $\{0, 1, \omega^<\}$ -LCA described in Example 5. There is a quantitative $\mathcal{C}wF$ where $\mathcal{C} = \mathcal{S}et$ and $\mathcal{L} = \mathcal{A}sm(\mathcal{A})$ the category of (modified) assemblies over \mathcal{A} , with $U(\Gamma) = |\Gamma|$. Types over Δ are given by Δ -indexed families of assemblies. Unresourced terms are given by set-theoretic functions, whereas resourced terms are realisable functions. Scaling of assemblies is defined by $\rho(\Gamma) = (|\Gamma|, \vDash_{\rho\Gamma})$ where $a \vDash_{\rho\Gamma} \gamma$ if there exists $b \in \mathcal{A}$, such that $a = !_{\rho}b$ and $b \vDash_{\Gamma} \gamma$. Similarly context addition is defined by the realisability relation where $a \vDash_{\Gamma_1 + \Gamma_2} \gamma$ if there exist $b, c \in \mathcal{A}$, such that $a = [b, c]$ with $b \vDash_{\Gamma_1} \gamma$ and $c \vDash_{\Gamma_2} \gamma$.*

Proof. Atkey [6, §4.2] shows that the given data forms a quantitative category with families interpreting dependent tensor types, Booleans, and dependent function types, where the types are interpreted as follows: for dependent tensor types, we have $|(x : S) \otimes T| = \{(s, t) \mid s \in |S|, t \in |T(s)|\}$, with $a \vDash_{(x:S)\otimes T} (s, t)$ if there exists b, c such that $a = [!_{\rho}b, c]$, $b \vDash_S s$, and $c \vDash_{T(s)} t$. Booleans are interpreted using that they are supported in \mathcal{A} . Finally dependent function types are interpreted by $|(x : S) \rightarrow T| = \prod_{x \in |S|} |T(s)|$ where $a \vDash_{(x:S)\rightarrow T} f$ if $b \vDash_S s$ implies $a \cdot !_{\rho}b \vDash_{T(s)} f(s)$.¹ We briefly sketch the interpretation for the remaining type formers we have introduced: We have $|\mathbf{I}| = |\top| = \{\star\}$ and $|\mathbf{0}| = \emptyset$, with $I \vDash_{\mathbf{I}} \star$ and $x \vDash_{\top} \star$ for any $x \in \mathcal{A}$. Further we have $|(x : S) \& T| = \{(s, t) \mid s \in |S|, t \in |T(s)|\}$ and $|S \oplus T| = |S| + |T|$, with $\mathit{case}(x, y) \vDash_{(x:S)\&T} (s, t)$ if $x \vDash_S s$ and $y \vDash_{T(s)} t$, and $[T, x] \vDash_{S \oplus T} \mathit{inl} s$ if $x \vDash_S s$, and similarly $[F, y] \vDash_{S \oplus T} \mathit{inr} t$ if $y \vDash_T t$ – for $S \& T$, the realisers for the projections can choose which realiser of x and y they want, and dually for $S \oplus T$, the realisers for the injections tag themselves with a Boolean T or F so that the realiser for the eliminator knows which case it is in. Finally the extensional identity type is interpreted as a subsingleton $|\mathit{Id}(s, t)| = \{\star \mid [s] = [t]\}$, with a trivial realiser $I \vDash_{\mathit{Id}(s,t)} \star$. ◀

We will use the above model to show that our proposed induction principles for data types are sound, by interpreting them in the model.

3 Data Types in Ordinary Type Theory

In this section, we recall how data types are usually presented in ordinary type theory.

3.1 Initial Algebra Semantics

An F -algebra for an endofunctor $F : \mathcal{C} \rightarrow \mathcal{C}$ is a pair (A, a) , where A is an object of \mathcal{C} and $a : F(A) \rightarrow A$ is a \mathcal{C} -morphism. A morphism between F -algebras (A, a) and (B, b) is a map $f : A \rightarrow B$ in \mathcal{C} , such that $f \circ a = b \circ F(f)$, i.e., the following diagram commutes:

$$\begin{array}{ccc} F(A) & \xrightarrow{a} & A \\ \downarrow F(f) & & \downarrow f \\ F(B) & \xrightarrow{b} & B \end{array}$$

Identity morphisms in \mathcal{C} are F -algebra morphisms, and F -algebra morphisms compose. Hence F -algebras and their morphisms form a category. An initial F -algebra is an initial object in this category, that is, it is an F -algebra (A, a) with a unique morphism fold_B to every F -algebra B . We often write μF or $\mu X.F(X)$ for the initial F -algebra.

¹ Unfortunately the proof in Atkey [6, §4.2] overlooked the need to also allow potentially unrealised elements in $|(x : S) \rightarrow T|$, which is needed to achieve the curry-uncurry isomorphism for unresourced terms in Atkey [6, Def. 3.5].

10:10 Quantitative Polynomial Functors

Initial F -algebras model inductively defined data types: the algebra map $a : F(A) \rightarrow A$ represents the constructors of the data type, and the unique morphism $\text{fold}_B : A \rightarrow B$ gives an “iteration principle” for defining functions out of the data type, à la pattern matching definitions – as we will see later, this “non-dependent” elimination rule together with uniqueness is actually equivalent to full dependent elimination for many functors F . This equivalence makes crucial use of the extensional identity type.

Concretely, working in the category of types and functions, we can build up many data types as initial algebras of *polynomial functors*, inductively generated by the following grammar:

$$F, G ::= \text{Id} \mid \text{Const}_A \mid F \times G \mid F + G \mid A \rightarrow F \quad (1)$$

where Id is the identity functor, $\text{Const}_A(X) = A$, and $\times, +, A \rightarrow F$ are defined pointwise. For example, the data type of natural numbers is the initial algebra of $\text{Const}_1 + \text{Id}$, and the data type of binary trees with elements of A stored at the leaves is the initial algebra of the functor $A + \text{Id} \times \text{Id}$. Note that all types generated by (1) are strictly positive, in the sense that no input variable occurs to the left of an arrow. This is important for initial algebras to exist.

3.2 Containers

Containers are based on a shapes-and-positions metaphor for data types, meant to represent how concrete data are stored at locations in memory. For example, every list $\ell : \text{List } X$ can be uniquely represented by a natural number $n : \mathbb{N}$ (given by the length of the list), together with a function $f : \text{Fin } n \rightarrow X$, which returns the elements of the list at the given position. The number $n : \mathbb{N}$ represents the *shape* of the list, and $\text{Fin } n$ is the type of *positions* of the given shape. In general, we have:

► **Definition 8** (Abbott, Altenkirch and Ghani [1]). *A container $S \triangleleft P$ is given by a type $S : \text{Type}$ of shapes, and a type family $P : S \rightarrow \text{Type}$ of positions. Its extension is given by the operation $\llbracket S \triangleleft P \rrbracket : \text{Type} \rightarrow \text{Type}$ defined by*

$$\llbracket S \triangleleft P \rrbracket X = (s : S) \times (P(s) \rightarrow X) .$$

More generally, containers can be presented in arbitrary locally Cartesian closed categories, but we restrict ourselves to the category of types here. The extension of a container formalises the intuition that an instantiation of the container is given by choosing a shape, and then a payload for each position. This action is functorial.

► **Proposition 9.** *Let $S \triangleleft P$ be a container. The extension $\llbracket S \triangleleft P \rrbracket$ extends to a functor $\text{Type} \rightarrow \text{Type}$, defined by $\llbracket S \triangleleft P \rrbracket(g)(s, f) = (s, g \circ f)$.*

The action on morphisms evidently preserves identities and composition. We call any functor isomorphic to one of the form $\llbracket S \triangleleft P \rrbracket$ a container functor. Container functors are closed under many type formers and operations, for example:

$$\begin{aligned} \text{Id} &\cong \llbracket \mathbf{1} \triangleleft (\lambda. \mathbf{1}) \rrbracket \\ \text{Const}_A &\cong \llbracket A \triangleleft (\lambda. \mathbf{0}) \rrbracket \\ \llbracket S \triangleleft P \rrbracket \times \llbracket S' \triangleleft P' \rrbracket &\cong \llbracket (S \times S') \triangleleft (\lambda(s, s'). P(s) + P'(s')) \rrbracket \\ \llbracket S \triangleleft P \rrbracket + \llbracket S' \triangleleft P' \rrbracket &\cong \llbracket (S + S') \triangleleft [P, P'] \rrbracket \\ A \rightarrow \llbracket S \triangleleft P \rrbracket &\cong \llbracket (A \rightarrow S) \triangleleft (\lambda f. (a : A) \times P(f(a))) \rrbracket \end{aligned} \quad (2)$$

Using the above isomorphisms, one can show that any polynomial functor in the sense of (1) can be represented as a container.

► **Theorem 10** (Dybjer [15]). *In extensional type theory, for any polynomial functor F , there is a container $S_F \triangleleft P_F$ such that $F \cong \llbracket S_F \triangleleft P_F \rrbracket$.*

Hence in extensional type theory, the problem of constructing initial algebras for polynomial functors reduces to the problem of constructing initial algebras of containers, which are given by Martin L of's W-type². As a consequence, we can restrict our attention to containers, that are easier to work with compared to polynomial functors, since they are not inductively generated.

4 Quantitative Containers

Based on their success for data types in ordinary type theory, it seems reasonable to try to generalise containers to the quantitative setting. Since categorical models of QTT are monoidal closed rather than Cartesian closed, they are in particular not locally Cartesian closed, and so, this is not a question of interpreting containers directly.

4.1 Quantitative Container Functors on the Category of Closed Types and Linear Functions

Working internally in QTT, we can keep the same notion of a container $S \triangleleft P$ as given by $S : \text{Type}$ and $P : S \rightarrow \text{Type}$. Since types are checked in the 0-fragment, we do not need to worry about linear uses of $s : S$ when forming $P(s) : \text{Type}$. However from now on, we change the extension of the container from using dependent pairs and functions, to using dependent tensors and linear functions respectively:

► **Definition 11.** *A quantitative container $S \triangleleft P$ is given by a QTT type $S : \text{Type}$ of shapes, and a QTT type family $P : S \rightarrow \text{Type}$ of positions. Its extension is given by the operation $\llbracket S \triangleleft P \rrbracket : \text{Type} \rightarrow \text{Type}$ defined by*

$$\llbracket S \triangleleft P \rrbracket X = (s : S) \otimes (P(s) \multimap X) .$$

As expected, container extensions are still functorial, if we move to the category Type_{Lin} of closed types and *linear* functions: a morphism from a type X to a type Y is given by a function $\vdash f : X \multimap Y$.

► **Proposition 12.** *Let $S \triangleleft P$ be a container. The extension $\llbracket S \triangleleft P \rrbracket$ extends to a functor $\text{Type}_{\text{Lin}} \rightarrow \text{Type}_{\text{Lin}}$, defined by $\llbracket S \triangleleft P \rrbracket(g) x = \text{let } (s, f) = x \text{ in } (s, g \circ f)$.*

The above proposition can be generalised to a category of types and functions over an arbitrary, fixed context of the shape $\Gamma = 0\Gamma$, i.e. a context where all variables are annotated with 0.

► **Example 13.** We can define the list container $\mathbb{N} \triangleleft \text{Fin}$ as before, but note that its use is rather complicated, due to linearity constraints: to define a function out of $\llbracket \mathbb{N} \triangleleft \text{Fin} \rrbracket X$, we have to make use of both the length n and the payload function $f : \text{Fin}(n) \rightarrow X$

² Recently, Hugunin [25] showed how this result can be extended also to intensional type theory.

10:12 Quantitative Polynomial Functors

exactly once. As an example, consider a slightly unusual **Maybe** container, given by $M = \mathbf{2} \triangleleft (\lambda b. \text{if } b \text{ then } \mathbf{I} \text{ else } \text{Fin } 0)$: we have two shapes, one with trivial positions (representing a **just** value), and one with $\text{Fin } 0$ positions (representing **nothing**). Now we can write a function $\text{head} : \llbracket \mathbb{N} \triangleleft \text{Fin} \rrbracket X \xrightarrow{1} \llbracket M \rrbracket X$, which given a tensor tuple (n, f) first inspects the length n ; if it is non-zero, $f : \text{Fin}(n') \xrightarrow{1} X$ and we can return $\text{just}(f\ 0)$, otherwise if $n = 0$ we can return $f : \text{Fin } 0 \xrightarrow{1} X$ itself as the payload function in the **nothing** case, thus using both the length and the payload of the list. However many other plausibly linear operations on lists can not be written with this representation – as we will see, this hints at a deficiency of the container representation of data types in a quantitative setting.

► **Example 14.** Hancock and Hyvernat [21] suggest to think of a container $Q \triangleleft A$ as an interaction structure: the shapes Q represent questions, or commands, and the positions A represent answers, or responses. An element of $\llbracket Q \triangleleft A \rrbracket X$ is thus a tuple (q, f) where q is a question, and $f : A(q) \xrightarrow{1} X$ is a linear function ready to give an element of X for every answer to q . An initial algebra $D = \mu X. \llbracket Q \triangleleft A \rrbracket X$ of $\llbracket Q \triangleleft A \rrbracket$ thus intuitively consists of wellfounded “dialogue trees” (q, f) where $q : Q$ is a question, and $f : A(q) \xrightarrow{1} D$ is a function ready to supply a dialogue subtree for each possible answer to the question. Linearity here means that to consume such a dialogue tree, we firstly need to consume the question, and then we get to choose *exactly one* answer to the question to continue the dialogue, and so on, until we reach a question with no answers. Similar to when modelling stateful operations [33], linearity prevents us from “going back in time” and speculatively trying a different answer. Dually, when constructing such a dialogue tree, the typing rules allow us to use the same resources when constructing different subtrees – only one subtree is eventually going to be explored anyway.

4.2 Closure Under Type Formers

Unfortunately, many of the isomorphisms in (2) do not carry over from ordinary containers to quantitative containers, due to the bifurcation into additive and multiplicative type formers in the linear setting.

► **Theorem 15.** *The identity functor is a quantitative container, but constant functors are not, in general.*

Proof. The identity functor is represented by the quantitative container $\mathbf{I} \triangleleft (\lambda. \mathbf{I})$, since $\mathbf{I} \otimes A \cong A$ and $\mathbf{I} \xrightarrow{1} A \cong A$ by Lemma 3. For showing that quantitative containers are not closed under constant functors, note that every model of ordinary type theory is also a model of QTT, with vacuous resource tracking [6, Prop 3.3]. Hence if $\text{Const}_A \cong \llbracket S \triangleleft P \rrbracket$, the same isomorphism must hold if all linear type formers are replaced by their Cartesian variants in the definition of S and P , because that would be the interpretation of the isomorphism in a vacuous model. Further, since the extension functor $\llbracket - \rrbracket$ is full and faithful for ordinary containers [1, Thm. 3.4], if $\text{Const}_A \cong \llbracket S \triangleleft P \rrbracket$ in a vacuous model, then in that model we must have $S \cong A$ and $(P(s) \rightarrow X) \cong \mathbf{1}$ by (2). Going back to the linear world, we must hence for each $s : S$ and type X have $(P(s) \xrightarrow{1} X) \cong \mathbf{I}$. In particular, for $X = \mathbf{0}$, there is a function of type $P(s) \xrightarrow{1} \mathbf{0}$, which implies $P(s) \cong \mathbf{0}$ since $\mathbf{0}$ is initial. But this absurd: since $(\mathbf{0} \xrightarrow{1} X) \cong \top$, we then would have $\top \cong \mathbf{I}$, but this isomorphism does not hold in all models. Hence there cannot be such a quantitative container $S \triangleleft P$. ◀

It is not hard to see that quantitative containers are closed under \oplus , as the isomorphism for $+$ in (2) is already linear. However the isomorphisms for closure under all the other type formers are not, and so quantitative containers are not closed under function spaces, \otimes , or $\&$. This is quite a blow for their usefulness as a general framework for data types – as a consequence, we will consider alternatives in Section 5.

4.3 Elimination Rules and Induction Principles

As we have seen in Example 14, it is sometimes helpful to consider initial algebras W of quantitative container functors $\llbracket S \triangleleft P \rrbracket$. The algebra map $c : \llbracket S \triangleleft P \rrbracket(W) \xrightarrow{1} W$ corresponds to the introduction rule of the type, and the mediating morphism into any other $\llbracket S \triangleleft P \rrbracket$ -algebra $B : \text{Type}$ corresponds to an elimination rule:

$$\frac{\vdash b : ((s : S) \otimes (P(s) \xrightarrow{1} B)) \xrightarrow{1} B}{\vdash \text{fold}_{(B,b)} : W \xrightarrow{1} B}$$

with computation rule $\text{fold}_{(B,b)}(c(x)) = b(\llbracket S \triangleleft P \rrbracket(\text{fold}_{(B,b)}) x)$ given by the fact that $\text{fold}_{(B,b)}$ is an algebra morphism. A priori, this only gives a *non-dependent* elimination rule (also known as a recursion principle), but by exploiting the uniqueness of the mediating map, we can also derive the following *dependent* elimination rule (induction principle) for any $\vdash Q : W \rightarrow \text{Type}$:

$$\frac{\vdash m : (s : S) \rightarrow (h : P(s) \xrightarrow{1} W) \rightarrow ((p : P(s)) \rightarrow Q[h(p)]) \xrightarrow{1} Q[c(s, h)]}{\vdash \text{elim}(Q, m) : (x : W) \rightarrow Q[x]} \quad (3)$$

This rule says that to prove $Q[x]$ for every $x : W$ using one copy of x , it is enough to prove $Q[c(s, h)]$ for $s : S$ and $h : P(s) \xrightarrow{1} W$, assuming $Q[h(p)]$ already holds for every $p : P(s)$, using one copy of s and the induction hypothesis, but zero copies of h . A priori, it is perhaps not completely obvious that this is the right usage annotation of the variables involved. Indeed, a noteworthy feature of Theorem 16 below is that these annotations naturally fall out from its proof, so that we can use it to derive the form of the induction principle in a principled manner in the quantitative setting.

In fact, just like in ordinary type theory, initiality and induction are actually equivalent principles: An algebra is initial if and only if it supports induction. This result and construction seems to have been discovered several times by several authors, for example Dybjer and Setzer [16], Hermida and Jacobs [23], and Awodey, Gambino and Sojakova [7]. Our contribution here is to adapt the proof to also account for linearity.

► **Theorem 16.** *Let $(W, c : \llbracket S \triangleleft P \rrbracket(W) \xrightarrow{1} W)$ be an $\llbracket S \triangleleft P \rrbracket$ -algebra. The algebra W is initial if and only if there is a term $\text{elim}(Q, m)$ as in (3) for every $w : W \vdash Q : \text{Type}$, satisfying the computation rule $\text{elim}(Q, c(x)) = \text{let } (s, h) = x \text{ in } m \text{ s } h (\lambda p. \text{elim}(Q, h(p)))$.*

Proof. Assuming that (W, c) is initial and the premises of the elimination rule, we build an $\llbracket S \triangleleft P \rrbracket$ -algebra on the dependent tensor type $(w : W) \otimes Q$, and get a unique mediating morphism $\text{fold} : W \xrightarrow{1} (w : W) \otimes Q$ by initiality. We compose with the second projection $\text{snd} : (x : (w : W) \otimes Q) \rightarrow Q[\text{fst}(x)]$ to get a map $(x : W) \rightarrow Q[\text{fst}(\text{fold}(x))]$. Note that the use of second projection is admissible due to the annotation of the first component $w : W$ – we are free to dispose of w , since it is used 0 times. To show that $\text{snd} \circ \text{fold}$ has the right type, we need to show that $Q[\text{fst}(\text{fold}(x))] = Q[x]$ for every $x : W$, but as this is a type equality, unrestricted use of terms is permissible. The map $\text{fst} : (w : W) \otimes Q \xrightarrow{1} W$ is an $\llbracket S \triangleleft P \rrbracket$ -algebra morphism, and thus the composite $\text{fst} \circ \text{fold} : W \xrightarrow{1} W$ is also one:

10:14 Quantitative Polynomial Functors

$$\begin{array}{ccc}
 \llbracket S \triangleleft P \rrbracket(W) & \xrightarrow{c} & W \\
 \downarrow & & \downarrow \text{fold} \\
 \llbracket S \triangleleft P \rrbracket((w \overset{0}{:} W) \otimes Q) & \longrightarrow & (w \overset{0}{:} W) \otimes Q \\
 \downarrow & & \downarrow \text{fst} \\
 \llbracket S \triangleleft P \rrbracket(W) & \xrightarrow{c} & W
 \end{array}$$

Thus $\text{fst} \circ \text{fold} = \text{id}$ holds by uniqueness of the mediating morphism out of W . The computation rule is exactly that fold is an $\llbracket S \triangleleft P \rrbracket$ -algebra morphism.

For the converse direction, assume that the elimination rule holds for (W, c) . Let $(A, f : \llbracket S \triangleleft P \rrbracket(A) \rightarrow A)$ be an arbitrary $\llbracket S \triangleleft P \rrbracket$ -algebra and define the type $Q[x] := A$ for all $x : W$. Hence the type of the method in the elimination simplifies to

$$m \overset{1}{:} (s \overset{1}{:} S) \rightarrow (h \overset{0}{:} P(s) \overset{1}{\rightarrow} W) \rightarrow ((p \overset{1}{:} P(s)) \rightarrow A) \overset{1}{\rightarrow} A$$

which we can define by $m = \lambda s. \lambda h. \lambda y. f(s, y)$, since we have 0 copies of h , and thus do not need to use it. The elimination principle gives a function $\text{fold} := \text{elim}(A, m) : W \overset{1}{\rightarrow} A$, which by the computation rule is an $\llbracket S \triangleleft P \rrbracket$ -algebra morphism. We show uniqueness of fold as follows: given another $\llbracket S \triangleleft P \rrbracket$ -algebra morphism $g : W \overset{1}{\rightarrow} A$, we use the elimination principle and equality reflection to prove $g(x) = \text{fold}(x)$ for every $x : W$. For x of the form $c(s, f)$, this follows linearly from both g and fold being $\llbracket S \triangleleft P \rrbracket$ -algebra morphisms and the induction hypothesis, and the result follows by equality reflection. \blacktriangleleft

5 Quantitative Polynomial Functors

As we have seen, quantitative containers give a seemingly well behaved notion of data types in quantitative type theory: they are functorial, and their algebras support induction principles if and only if they are initial, which is a property that is usually easier to verify in models. However, there is a caveat – most quantitative versions of standard data types are not quantitative containers in the above sense, because quantitative containers are not closed under many type formers, as discussed in Section 4.2.

Consider, for example, the natural numbers, the initial algebra of the polynomial functor $F(X) = \mathbf{1} + X$, or binary trees, the initial algebra of $G(X) = A + X \times X$. Their representations as initial algebras of containers in Theorem 10 crucially depend on the isomorphisms $(\mathbf{0} \rightarrow X) \cong \mathbf{1}$ and $(\mathbf{2} \rightarrow X) \cong X \times X$, respectively. If we consider the corresponding QTT data types given by $F(X) = \mathbf{I} \oplus X$ and $G(X) = A \oplus (X \otimes X)$, we see using Lemma 3 that the QTT counterparts of the above isomorphisms do not hold: $(\mathbf{0} \overset{1}{\rightarrow} X) \cong \top \not\cong \mathbf{I}$ and $(\mathbf{2} \overset{1}{\rightarrow} X) \cong X \& X \not\cong X \otimes X$.

5.1 A Grammar for Quantitative Polynomial Functors

Since we can no longer reduce all data types we are interested in to quantitative containers, we instead resort to generating them inductively. This follows the same pattern as for polynomial functors in the grammar (1), except that we now have multiplicative and additive versions of many type formers.

► **Definition 17.** *The class of quantitative polynomial type expressions is inductively generated by the following grammar:*

$$F, G ::= \text{Id} \mid \text{Const}_A \mid F \otimes G \mid F \oplus G \mid F \& G \mid A \xrightarrow{1} F$$

If the type expression is generated without making use of the $A \xrightarrow{1} F$ production, we call it finitary.

A simple induction on how the type expression is generated proves that all quantitative polynomial type expressions are functors of type $\text{Type}_{\text{Lin}} \rightarrow \text{Type}_{\text{Lin}}$. Hence we are justified in calling them polynomial functors.

► **Example 18.** A functor describing the natural numbers is given by the finitary type expression $F = \text{Const}_{\mathbf{1}} \oplus \text{Id}$. An element of $F(X)$ is either $\text{inl } \star$, representing 0, or of the form $\text{inr } n$, representing the successor of n . A function $F(X) \xrightarrow{1} A$ uses no resources for the zero case, and gets one copy of n in the successor of n case. This is different from an attempted quantitative container representation of F , which would be given by $\mathbf{2} \triangleleft P$, where $P(\text{true}) = \mathbf{0}$ and $P(\text{false}) = \mathbf{1}$, since the representation of 0 there would also have a “junk” term of type $(\mathbf{0} \xrightarrow{1} X) \cong \top$ around.

► **Example 19.** Similarly, a finitary polynomial functor describing binary trees is given by $G = \text{Const}_A \oplus (\text{Id} \otimes \text{Id})$. An element is either of the form $\text{inl } a$ for some $a : A$, representing a leaf with label a , or of the form $\text{inr } (\ell, r)$, representing a node with subtrees ℓ and r . A function $G(X) \xrightarrow{1} B$ must be able to deal with either getting one copy of $a : A$, or one copy each of subtrees ℓ and r . In contrast, an attempted quantitative container representation of G would be given by $(A \oplus \mathbf{1}) \triangleleft P$, where $P(\text{inl } a) = \mathbf{0}$ and $P(\text{inr } \star) = \mathbf{2}$. This would have the same “junk term” problem for leaves as zero does in Example 18, but in addition, a function $\llbracket (A \oplus \mathbf{1}) \triangleleft P \rrbracket \xrightarrow{1} B$ would only get access to *one* of the two subtrees ℓ and r . If this is the intended use case, like in Example 14, then one can instead change the quantitative polynomial functor to $G' = \text{Const}_A \oplus (\text{Id} \& \text{Id})$.

5.2 Elimination Rules and Induction Principles

In order to consider initial algebras, a functor is all we need, but to formulate induction principles, we now need to construct some additional machinery: we need to explain what the type of induction hypothesis is for a given functor, and what the computation rule should be. For quantitative containers, we could do this directly, but since quantitative polynomial functors are inductively generated, we also need to proceed inductively. Our main tool is the predicate lifting [23] of polynomial functors, which will be used as the type of induction hypothesis.

10:16 Quantitative Polynomial Functors

► **Definition 20.** Let F be a quantitative polynomial functor. We define its predicate lifting $\widehat{F} : (Q : X \rightarrow \text{Type}) \rightarrow (F(X) \rightarrow \text{Type})$ by induction on F :

$$\begin{aligned} \widehat{\text{Id}}(Q, x) &= Q(x) \\ \widehat{\text{Const}}_A(Q, x) &= (a : A) \otimes (a = x) \\ \widehat{F \otimes G}(Q, x) &= \widehat{F}(Q, \text{proj}_1 x) \otimes \widehat{G}(Q, \text{proj}_2 x) \\ \widehat{F \oplus G}(Q, x) &= ((x_1 : F(X)) \otimes (p : x = \text{inl } x_1) \otimes \widehat{F}(Q, x_1)) \oplus \\ &\quad ((x_2 : G(X)) \otimes (p : x = \text{inr } x_2) \otimes \widehat{G}(Q, x_2)) \\ \widehat{F \& G}(Q, x) &= \widehat{F}(Q, \text{proj}_1 x) \& \widehat{G}(Q, \text{proj}_2 x) \\ \widehat{A \xrightarrow{1} F}(Q, x) &= (a : A) \rightarrow \widehat{F}(Q, x(a)). \end{aligned}$$

The lifting follows the same structure as the underlying functor. Note that we can use projections $\text{proj}_i := \lambda x. \text{let } (x_1, x_2) = x \text{ in } x_i$ in the $F \otimes G$ case, as we are defining a type, and hence we are in the 0 fragment. The only possibly surprising case is the lifting of constant functors $\widehat{\text{Const}}_A(Q, x) = (a : A) \otimes (a = x)$; in a non-quantitative setting, this would be a complicated way to define $\widehat{\text{Const}}_A(Q, x) = \mathbf{1}$, since, in the language of homotopy type theory, singletons $(a : A) \times (a = x)$ are contractible [36, Lem. 3.11.8]. However the point is that we get one “extra” copy a of x which can be used even when we have zero copies of x available. This is crucial for the proof of Lemma 24 below.

We will use predicate liftings to formulate the notion of induction hypothesis for F -algebras for a quantitative polynomial functor F . For formulating computation rules, we will furthermore make use of the following lemma, which states that each predicate lifting has a “functorial action” on *dependent* functions:

► **Lemma 21.** Let F be a quantitative polynomial functor. If $f : (x : X) \rightarrow Q(x)$ then we can define $\widehat{F}(f) : (y : F(X)) \rightarrow \widehat{F}(Q, y)$. In addition, if $g : Y \xrightarrow{1} X$, then $\widehat{F}(Q, F(g)(y)) = \widehat{F}(Q \circ g, y)$, and $\widehat{F}(f \circ g) = \widehat{F}(f) \circ F(g)$.

The proof is again a simple induction on the buildup of F . Since there are no identity dependent functions or compositions of dependent functions in general for this fixed form, it does not make sense to ask for \widehat{F} to preserve neither identities or composition in general, beyond the “mixed” \widehat{F} - F preservation stated in the lemma. However Lemma 25 below implies that \widehat{F} does preserve identities and compositions when these make sense.

We now have all the ingredients we need to define what it means for an algebra to support elimination and computation rules. Let F be a quantitative polynomial functor, and (W, c) an F -algebra. The dependent elimination rule (or induction principle) for W , for any predicate $\vdash Q : W \rightarrow \text{Type}$, is stated as follows:

$$\frac{\vdash m : (y : F(W)) \rightarrow \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))}{\vdash \text{elim}(Q, m) : (x : W) \rightarrow Q(x)} \quad (4)$$

with computation rule $\text{elim}(Q, m)(c(y)) = m y (\widehat{F}(\text{elim}(Q, m), y))$. Note how the computation rule is making use of the action of \widehat{F} on dependent functions from Lemma 21.

► **Example 22.** Recall from Example 18 that the quantitative polynomial functor $F = \text{Const}_{\mathbf{1}} \oplus \text{Id}$ describes the data type of natural numbers. The type of induction hypothesis for zero $:= c(\text{inl } \star)$ is, up to isomorphism, $\widehat{F}(Q, \text{inl } \star) \cong \mathbf{1}$, i.e., it contains no information, and the induction hypothesis for $\text{succ } n := c(\text{inr } n)$ is isomorphic to $\widehat{F}(Q, \text{inr } n) \cong \widehat{\text{Id}}(Q, n) = Q(n)$, as expected. We would thus expect the induction principle to be familiarly stated as follows

$$\frac{\vdash m_z : Q(\text{zero}) \quad \vdash m_s : (n : \mathbb{N}^0) \rightarrow Q(n) \xrightarrow{1} Q(\text{suc } n)}{\vdash \text{elim}(Q, m) : (x : \mathbb{N}^1) \rightarrow Q(x)}$$

and indeed it can be, up to isomorphism. An important point is that when translating between $m : (y : F(W)) \rightarrow \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))$ and m_z and m_s as above, we cannot make case distinctions on $y : F(W)$, since we have zero copies of y available. Instead, we have to split on the $\widehat{F}(Q, y)$ argument, which in turn will refine y . This is why the definition of $\widehat{F} \oplus \widehat{G}(Q, x)$ is designed the way it is, rather than just giving cases for $\widehat{F} \oplus \widehat{G}(Q, \text{inl } x_1)$ and $\widehat{F} \oplus \widehat{G}(Q, \text{inr } x_2)$ directly.

► **Example 23.** The quantitative polynomial functor $G = \text{Const}_A \oplus (\text{Id} \otimes \text{Id})$ from Example 19 describes binary trees. For leaves, we have $\widehat{G}(Q, \text{inl } x) \cong (a : A) \otimes (a = x)$, and for nodes we have $\widehat{G}(Q, \text{inr } x) \cong Q(\text{proj}_1 x) \otimes Q(\text{proj}_2 x)$. Hence the induction principle becomes

$$\frac{\vdash m_l : (a : A) \rightarrow Q(\text{leaf } a) \quad \vdash m_n : (\ell : \text{Tree}_A) \rightarrow (r : \text{Tree}_A) \rightarrow Q(\ell) \xrightarrow{1} Q(r) \xrightarrow{1} Q(\text{node } \ell r)}{\vdash \text{elim}(Q, m) : (t : \text{Tree}_A) \rightarrow Q(t)}$$

Note how the method m_l gets one rather than zero copies of $a : A$, thanks to the definition of $\widehat{G}(Q, \text{inl } x) \cong (a : A) \otimes (a = x)$.

We now aim to show that initiality and induction are equivalent also for quantitative polynomial functors. The proof follows the same pattern as Theorem 16, but we need some additional lemmas. Firstly, we need that F distributes over dependent tensor products of the form $(w : W) \otimes Q$ in an appropriate sense:

► **Lemma 24.** *For a quantitative polynomial functor F , we have*

$$\text{dist}_F : F((x : W) \otimes Q(x)) \xrightarrow{1} (y : F(W)) \otimes \widehat{F}(Q, y)$$

with $\text{proj}_1 \circ \text{dist}_F = F(\text{proj}_1)$ as an equation in the 0-fragment.

Secondly, for deriving initiality from induction, we need that \widehat{F} basically reduces to F for constant predicates:

► **Lemma 25.** *If $Q(x) = A$ for every $x : X$, i.e., Q is constant, then $\widehat{F}(Q, y) \cong F(A)$ for every $y : F(X)$, and $\widehat{F}(f) \cong F(f)$ for every $f : X \xrightarrow{1} A$.*

Armed with these lemmas, we can now attack our main theorem:

► **Theorem 26.** *Let F be a quantitative polynomial functor and $(W, c : F(W) \xrightarrow{1} W)$ an F -algebra. The algebra W is initial if and only if there is a term $\text{elim}(Q, m)$ as in (4) for every $w : W \vdash Q : \text{Type}$, satisfying the computation rule $\text{elim}(Q, m)(c(y)) = m y (\widehat{F}(\text{elim}(Q, m), y))$.*

Proof. Assuming that (W, c) is initial, and given $m : (y : F(W)) \rightarrow \widehat{F}(Q, y) \xrightarrow{1} Q(c(y))$, we construct an F -algebra

$$F((x : W) \otimes Q(x)) \xrightarrow{\text{dist}_F} (y : F(W)) \otimes \widehat{F}(Q, y) \xrightarrow{\langle c, m \rangle} (x : W) \otimes Q(x)$$

10:18 Quantitative Polynomial Functors

and by initiality, we get $\text{fold}_{\langle c, m \rangle \circ \text{dist}_F} : W \xrightarrow{1} (x \overset{0}{:} W) \otimes Q(x)$. By Lemma 24, the following diagram commutes, meaning that proj_1 is an algebra morphism from $(x \overset{0}{:} W) \otimes Q(x)$ to W in the 0-fragment:

$$\begin{array}{ccccc} F((x \overset{0}{:} W) \otimes Q(x)) & \xrightarrow{\text{dist}_F} & (y \overset{0}{:} F(W)) \otimes \widehat{F}(Q, y) & \xrightarrow{\langle c, m \rangle} & (x \overset{0}{:} W) \otimes Q(x) \\ \downarrow F(\text{proj}_1) & & \downarrow \text{proj}_1 & & \downarrow \text{proj}_1 \\ F(W) & \xlongequal{\quad} & F(W) & \xrightarrow{c} & W \end{array}$$

Hence by uniqueness, we have $\text{proj}_1 \circ \text{fold}_{\langle c, m \rangle \circ \text{dist}_F} = \text{id}$, and hence we can define $\text{elim}(Q, m) = \text{proj}_2 \circ \text{fold}_{\langle c, m \rangle \circ \text{dist}_F} : (y \overset{1}{:} W) \rightarrow Q(y)$. The computation rule follows from the fact that fold is an F -algebra morphism, and the second part of Lemma 21. Conversely, using Lemma 25 we can use the induction principle on a constant predicate to get a morphism $W \xrightarrow{1} B$ for an F -algebra B . We prove uniqueness with another instantiation of the induction principle, together with function extensionality. \blacktriangleleft

5.3 Initial Algebras of Finitary Quantitative Polynomial Functors in the Realisability Model

Initial algebras of a polynomial functor $F : \mathcal{C} \rightarrow \mathcal{C}$ need not necessarily exist for an arbitrary category \mathcal{C} . When the category in question is $\mathcal{S}et$, however, the initial algebra μF always exists for polynomial functors. We show that a similar result holds for finitary quantitative polynomial functors in the quantitative CwF from Proposition 7. We refer to that category as the realisability model \mathcal{M}_r .

Semantic types in \mathcal{M}_r are interpreted by a collection of assemblies. Thus to construct the initial algebra of a quantitative polynomial functor F , we first need to define a type μF , endow it with a realisable structure map $c : F(\mu F) \rightarrow \mu F$, and finally show that the mediating morphism fold to any other F -algebra $(X, \alpha : F(X) \rightarrow X)$ is realisable. For simplicity of notation, we present the construction as in the empty context.

Let $\tilde{F} : \mathcal{S}et \rightarrow \mathcal{S}et$ designate the set functor obtained from F by replacing each linear connective with a Cartesian one. This is a polynomial functor, and hence has an initial algebra $(\mu \tilde{F}, \tilde{c} : \tilde{F}(\mu \tilde{F}) \rightarrow \mu \tilde{F})$. Our plan is to augment the \tilde{F} -initial algebra with realisability information following the structure of F . Denote by \bar{n} the encoding of the numeral n using tupling and Booleans in the underlying R -LCA $\mathcal{A} = \mathcal{P}(\mathbb{N})$ from Example 5.

► **Construction 27.** Let F be a quantitative polynomial functor and let $(\mu \tilde{F}, \tilde{c} : \tilde{F}(\mu \tilde{F}) \rightarrow \mu \tilde{F})$ be the initial algebra of \tilde{F} in $\mathcal{S}et$. We define $(\vDash_{\mu F} x) \subseteq \mathcal{A}$ by induction on $x \in \mu \tilde{F}$ and the buildup of F : it is sufficient to define $a \vDash_{\mu F} \tilde{c}(y)$ for some $y \in \tilde{F}(\mu \tilde{F})$, assuming we have already defined the relation $a' \vDash_{\mu F} z$ for all structurally smaller z .

- if $F = \text{Id}$, then $y \in |\mu F|$ and we have already defined $(\vDash_{\mu F} x) \subseteq \mathcal{A}$. We define $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A}, \text{ such that } b \vDash_{\mu F} y \wedge a = [\bar{1}, b]$.
- if $F = \text{Const}_A$, then $y \in |A|$ and we define $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A}, \text{ such that } b \vDash_A y \wedge a = [\bar{2}, b]$.
- if $F = F' \otimes G'$, then there are some $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = (y_1, y_2)$. Let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b_1, b_2 \in \mathcal{A}, \text{ such that } b_1 \vDash_{F'(\mu F)} y_1 \wedge b_2 \vDash_{G'(\mu F)} y_2 \wedge a = [\bar{3}, [b_1, b_2]]$
- if $F = F' \oplus G'$, there are $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = \text{inl}(y_1)$ or $y = \text{inr}(y_2)$.
 If $y = \text{inl}(y_1)$, let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b_1 \in \mathcal{A}, \text{ such that } b_1 \vDash_{F'(\mu F)} y_1 \wedge a = [\bar{4}, [T, b_1]]$;
 if $y = \text{inr}(y_2)$, let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b_2 \in \mathcal{A}, \text{ such that } b_2 \vDash_{G'(\mu F)} y_2 \wedge a = [\bar{4}, [F, b_2]]$.

- if $F = F' \& G'$, there are $y_1 \in |F'(\mu F)|$ and $y_2 \in |G'(\mu F)|$, such that $y = (y_1, y_2)$.
Let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b_1, b_2 \in \mathcal{A}$, such that $b_1 \vDash_{F'(\mu F)} y_1 \wedge b_2 \vDash_{G'(\mu F)} y_2 \wedge a = [\bar{5}, \text{case}(b_1, b_2)]$.
- if $F = A \xrightarrow{1} F'$, then $y \in |A \rightarrow F(\mu F)|$. Let $a \vDash_{\mu F} \tilde{c}(y) \iff \exists b \in \mathcal{A}$, s.t. $(\forall i \in A \forall a_i \in \mathcal{A}, a_i \vDash_A i \implies b \cdot a_i \vDash_{F'(\mu F)} y(i)) \wedge a = [\bar{6}, b]$.

The only perhaps slightly odd case is for the function space $F = A \xrightarrow{1} F'$, which is obtained through unwinding the definition of a realisable function. The following lemma is easily provable by induction on the buildup of F , using combinatory completeness to define the realisers.

► **Lemma 28.** *Let F be a quantitative polynomial functor and μF be constructed as in Construction 27. Then the function $c : F(\mu F) \rightarrow \mu F$ is realisable.*

Similarly, we can prove that the mediating morphism fold is realisable by induction on its argument $x \in \mu \tilde{F}$, when F is finitary, i.e., when the rule $A \xrightarrow{1} F'$ is omitted. This is making essential use of the fact that fold is an \tilde{F} -algebra morphism in $\mathcal{S}et$.

► **Lemma 29.** *Let F be a finitary quantitative polynomial functor, and $(X, \alpha : F(X) \rightarrow X)$ be an F -algebra. Then the map $\text{fold} : \mu F \rightarrow X$ is realisable.*

It is important to understand why we had to assume that F was finitary: given a function $y \in |A \xrightarrow{1} F'(\mu F)|$, s.t. $x = c(y)$, we get a family of realisers $\{a_i | a_i \vDash_{F'(\mu F)} \text{fold}(y(i))\}_{i \in |A|}$ by the induction hypothesis. However, we have to construct a realiser for $\text{fold}(c(y))$ itself – that is, we have to find some $b \in \mathcal{A}$, s.t. $\forall i \in |A|$ if $d \vDash_A i$, then $b \cdot d = a_i$. But there is no general construction for such a b , because there is no guarantee that the function $i \mapsto a_i$ is linear, or even computable.

Even though there might be elements without realisers in our assemblies in general, we can show by induction that elements in $|\mu F|$ have realisers, for finitary quantitative polynomial functors F , assuming of course that F is not constructed from Const_A for some A with elements without realisers. We have to restrict ourselves to finitary quantitative polynomial functors for the same reason as in the proof of Lemma 29: the induction hypothesis for $F = A \xrightarrow{1} F'$ only gives us a collection of realisers, but no computable function that selects one for every realiser of $a \in |A|$.

► **Proposition 30.** *Let F be a finitary quantitative polynomial functor such that if F was generated using a Const_A rule, then every $x \in |A|$ has a realiser. Then every element $x \in |\mu F|$ has a realiser.*

Altogether, we have now shown how to interpret the type μF as an assembly by making use of the initial algebra of \tilde{F} in $\mathcal{S}et$, and how both the algebra map $c : F(\mu F) \rightarrow \mu F$ and the unique mediating morphism $\text{fold} : \mu F \rightarrow X$ are realisable, the latter if F is finitary. Furthermore, if built from components where every element has a realiser, the initial algebra will retain this property, which is often important for reasoning about terms in the model. Hence initial algebras of finitary quantitative polynomial functors are supported in the realisability model \mathcal{M}_r .

6 Conclusions and Future Work

We have given the first principled account of data types in quantitative type theory. This is necessary, since many established facts about ordinary data types, such as the universal applicability of containers to represent all strictly positive type formers, do not carry over

to the quantitative setting. Instead we have considered quantitative polynomial functors inductively generated by a grammar. By firstly reducing elimination rules to initiality, and then concretely constructing initial algebras in the model, we have shown how finitary data types can be given semantics in a linear realisability model of QTT. This gives a precise mathematical meaning to a subset of the data types that can be defined in Idris 2, and reassurance that they are canonical, in the sense that they satisfy the universal property of initiality.

The equivalence between elimination rules and initiality works for arbitrary quantitative polynomial functors, but our proof of the existence of initial algebras in the realisability model is restricted to finitary quantitative polynomial functors. We conjecture that also infinitary data types are supported in this model, but this result is out of reach of our current proof method. Our data types are also functors on categories of *closed* types, and correspondingly we only derive induction principles in empty contexts; it would be good to relax this restriction.

Going beyond simple polynomial functors, we believe it should be possible to adapt Gambino and Hyland’s reduction of the existence of initial algebras of indexed containers to the existence of initial algebras of containers [18] to the quantitative setting, at least with an extensional identity type. It seems as if the proof should extend to quantitative polynomial functors as well, with some more work. However going further, Kaposi and Kovács have had great success describing and modelling quotient and higher inductive-inductive types using a notion of signature based on internal categories with families to model the intricate dependencies between different constructors [27, 28, 26]. It would be interesting to see if a similar approach of internal type theory [14] using *quantitative* categories with families could also work to describe more expressive data types in the setting of QTT.

References




- 1 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew D. Gordon, editor, *Foundations of Software Science and Computation Structures*, pages 23–38. Springer, 2003. doi:10.1007/3-540-36576-1_2.
- 2 Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers: Constructing strictly positive types. *Theoretical Computer Science*, 342(1):3–27, 2005. doi:10.1016/j.tcs.2005.06.002.
- 3 Andreas Abel and Jean-Philippe Bernardy. A unified view of modalities in type systems. *Proceedings of the ACM on Programming Languages*, 4(ICFP):1–28, 2020. doi:10.1145/3408972.
- 4 Samson Abramsky, Esfandiar Haghverdi, and Philip Scott. Geometry of interaction and linear combinatory algebras. *Mathematical Structures in Computer Science*, 12(5):625–665, 2002. doi:10.1017/S0960129502003730.
- 5 Thorsten Altenkirch, Paolo Capriotti, Gabe Dijkstra, Nicolai Kraus, and Fredrik Nordvall Forsberg. Quotient inductive-inductive types. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, Lecture Notes in Computer Science, pages 293–310. Springer, Heidelberg, 2018. doi:10.1007/978-3-319-89366-2_16.
- 6 Robert Atkey. Syntax and semantics of quantitative type theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '18*, pages 56–65. ACM Press, 2018. doi:10.1145/3209108.3209189.
- 7 Steve Awodey, Nicola Gambino, and Kristina Sojakova. Homotopy-initial algebras in type theory. *Journal of the ACM*, 63(6), 2017. doi:10.1145/3006383.
- 8 Edwin Brady. Idris 2: Quantitative Type Theory in Practice. In Anders Møller and Manu Sridharan, editors, *35th European Conference on Object-Oriented Programming (ECOOP 2021)*, volume 194 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 9:1–9:26. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPIcs.ECOOP.2021.9.

- 9 Iliano Cervesato and Frank Pfenning. A Linear Logical Framework. *Information and Computation*, 179(1):19–75, 2002. doi:10.1006/inco.2001.2951.
- 10 Pritam Choudhury, Harley Eades III, Richard A. Eisenberg, and Stephanie Weirich. A graded dependent type system with a usage-aware semantics. *Proc. ACM Program. Lang.*, 5(POPL):1–32, 2021. doi:10.1145/3434331.
- 11 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.
- 12 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 '18*, pages 255–264. Association for Computing Machinery, 2018. doi:10.1145/3209108.3209197.
- 13 Thierry Coquand and Christine Paulin. Inductively defined types. In Per Martin-Löf and Grigori Mints, editors, *COLOG-88*, pages 50–66. Springer, 1990. doi:10.1007/3-540-52335-9_47.
- 14 Peter Dybjer. Internal type theory. In *Lecture Notes in Computer Science*, volume 1158 LNCS, pages 120–134. Springer, Berlin, Heidelberg, 1996. doi:10.1007/3-540-61780-9_66.
- 15 Peter Dybjer. Representing inductively defined sets by wellorderings in Martin-Löf’s type theory. *Theoretical Computer Science*, 176(1):329–335, 1997. doi:10.1016/S0304-3975(96)00145-4.
- 16 Peter Dybjer and Anton Setzer. Induction–recursion and initial algebras. *Annals of Pure and Applied Logic*, 124(1):1–47, 2003. doi:10.1016/S0168-0072(02)00096-9.
- 17 Peng Fu, Kohei Kishida, and Peter Selinger. Linear Dependent Type Theory for Quantum Programming Languages: Extended Abstract. In *Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2020: Proceedings of the 35th Annual ACM/IEEE Symposium on Logic in Computer Science*, pages 440–453. Association for Computing Machinery, 2020. doi:10.1145/3373718.3394765.
- 18 Nicola Gambino and Martin Hyland. Wellfounded Trees and Dependent Polynomial Functors. In Stefano Berardi, Mario Coppo, and Ferruccio Damiani, editors, *Types for Proofs and Programs, TYPES 2003*, volume 3085 of *Lecture Notes in Computer Science*, pages 210–225. Springer, Berlin, Heidelberg, 2003. doi:10.1007/978-3-540-24849-1_14.
- 19 Nicola Gambino and Joachim Kock. Polynomial functors and polynomial monads. *Mathematical Proceedings of the Cambridge Philosophical Society*, 154(1):153–192, 2013. doi:10.1017/S0305004112000394.
- 20 Jean-Yves Girard. Linear logic. *Theoretical Computer Science*, 50(1):1–101, 1987. doi:10.1016/0304-3975(87)90045-4.
- 21 Peter Hancock and Pierre Hyvernat. Programming interfaces and basic topology. *Annals of Pure and Applied Logic*, 137(1-3):189–239, 2006. doi:10.1016/j.apal.2005.05.022.
- 22 Peter Hancock and Anton Setzer. Interactive Programs in Dependent Type Theory. In Peter Clote and Helmut Schwichtenberg, editors, *Computer Science Logic, CSL 2000*, volume 1862 of *Lecture Notes in Computer Science*, pages 317–331. Springer, Berlin, Heidelberg, 2000. doi:10.1007/3-540-44622-2_21.
- 23 Claudio Hermida and Bart Jacobs. Structural Induction and Coinduction in a Fibrational Setting. *Information and Computation*, 145(2):107–152, 1998. doi:10.1006/inco.1998.2725.
- 24 Naohiko Hoshino. Linear Realizability. In *Computer Science Logic*, volume 4646 LNCS, pages 420–434. Springer, Berlin, Heidelberg, Berlin, Heidelberg, 2007. doi:10.1007/978-3-540-74915-8_32.
- 25 Jasper Hugunin. Why not W? In *26th International Conference on Types for Proofs and Programs (TYPES 2020)*, 2020. doi:{10.4230/LIPIcs.TYPES.2020.8}.

- 26 Ambrus Kaposi and András Kovács. Signatures and induction principles for higher inductive-inductive types. *Log. Methods Comput. Sci.*, 16(1), 2020. doi:10.23638/LMCS-16(1:10)2020.
- 27 Ambrus Kaposi, András Kovács, and Thorsten Altenkirch. Constructing quotient inductive-inductive types. *Proc. ACM Program. Lang.*, 3(POPL):2:1–2:24, 2019. doi:10.1145/3290315.
- 28 András Kovács and Ambrus Kaposi. Large and infinitary quotient inductive-inductive types. In Holger Hermanns, Lijun Zhang, Naoki Kobayashi, and Dale Miller, editors, *LICS '20: 35th Annual ACM/IEEE Symposium on Logic in Computer Science, Saarbrücken, Germany, July 8–11, 2020*, pages 648–661. ACM, 2020. doi:10.1145/3373718.3394770.
- 29 Neelakantan R. Krishnaswami, Pierre Pradic, and Nick Benton. Integrating Linear and Dependent Types. *ACM SIGPLAN Notices*, 50(1):17–30, 2015. doi:10.1145/2775051.2676969.
- 30 Peter Lefanu Lumsdaine and Michael Shulman. Semantics of higher inductive types. *Mathematical Proceedings of the Cambridge Philosophical Society*, 169(1):159–208, 2020. doi:10.1017/S030500411900015X.
- 31 Conor McBride. I Got Plenty o’ Nuttin’. In *Lecture Notes in Computer Science*, volume 9600, pages 207–233. Springer Verlag, 2016. doi:10.1007/978-3-319-30936-1_12.
- 32 Dominic Orchard, Vilem-Benjamin Liepelt, and Harley Eades III. Quantitative program reasoning with graded modal types. *Proceedings of the ACM on Programming Languages*, 3(ICFP):110:1–110:30, 2019. doi:10.1145/3341714.
- 33 Uday S. Reddy. A linear logic model of state. Manuscript, 1993.
- 34 Jan M. Smith. The independence of Peano’s fourth axiom from Martin-Löf’s type theory without universes. *Journal of Symbolic Logic*, 53(3):840–845, 1988. doi:10.2307/2274575.
- 35 Tomáš Svoboda. Additive pairs in quantitative type theory. Master thesis, Charles University Prague, 2021. doi:20.500.11956/127263.
- 36 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. <https://homotopytypetheory.org/book>, Institute for Advanced Study, 2013.
- 37 Philip Wadler. Linear types can change the world! In M. Broy and C. Jones, editors, *Programming Concepts and Methods*. North Holland, 1990.

Types and Terms Translated: Unrestricted Resources in Encoding Functions as Processes

Joseph W. N. Paulus   
University of Groningen, The Netherlands

Daniele Nantes-Sobrinho   
University of Brasília, Brazil
Imperial College London, UK

Jorge A. Pérez   
University of Groningen, The Netherlands

Abstract

Type-preserving translations are effective rigorous tools in the study of core programming calculi. In this paper, we develop a new typed translation that connects sequential and concurrent calculi; it is governed by type systems that control *resource consumption*. Our main contribution is the source language, a new resource λ -calculus with non-collapsing non-determinism and failures, dubbed $u\lambda_{\oplus}^{\xi}$. In $u\lambda_{\oplus}^{\xi}$, resources are split into linear and unrestricted; failures are explicit and arise from this distinction. We define a type system based on intersection types to control resources and fail-prone computation. The target language is $s\pi$, an existing session-typed π -calculus that results from a Curry-Howard correspondence between linear logic and session types. Our typed translation subsumes our prior work; interestingly, it treats unrestricted resources in $u\lambda_{\oplus}^{\xi}$ as client-server session behaviours in $s\pi$.

2012 ACM Subject Classification Theory of computation \rightarrow Type structures; Theory of computation \rightarrow Process calculi

Keywords and phrases Resource λ -calculus, intersection types, session types, process calculi

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.11

Related Version Online appendix with omitted proofs and further examples:

Full Version: <http://arxiv.org/abs/2112.01593> [18]

Funding Research partially supported by the Dutch Research Council (NWO) under project No. 016.Vidi.189.046 (Unifying Correctness for Communicating Software).

Acknowledgements We are grateful to the anonymous reviewers for their constructive feedback.

1 Introduction

Context. *Type-preserving translations* are effective rigorous tools in the study of core programming calculi. They can be seen as an abstract counterpart to the type-preserving compilers that enable key optimisations in the implementation of programming languages. The goal of this paper is to develop a new typed translation that connects sequential and concurrent calculi, and is governed by type systems that control *resource consumption*.

A central idea in the resource λ -calculus is to consider that in an application $M N$ the argument N is a *resource* of possibly limited availability. This generalisation of the λ -calculus triggers many fascinating questions, such as typability, solvability, expressiveness power, etc., which have been studied in different settings (see, e.g., [1, 3, 16, 7]). In established resource λ -calculi, such as those by Boudol [1] and by Pagani and Ronchi della Rocca [16], a more general form of application is considered: a term can be applied to a bag of resources $B = \{N_1\} \cdot \dots \cdot \{N_k\}$, where N_1, \dots, N_k denote terms; then, an application $M B$ must take into account that each N_i may be reusable or not. Thus, non-determinism is natural in



© Joseph W. N. Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez;
licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 11; pp. 11:1–11:24

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

resource λ -calculi, because a term has now multiple ways of consuming resources from the bag. This bears a strong resemblance with process calculi such as the π -calculus [15], in which concurrent interactions are intrinsically non-deterministic.

There are different flavors of non-determinism. Over two decades ago, Boudol and Laneve [2, 3] explored connections between a resource λ -calculus and the π -calculus. In their setting, an application $M B$ would branch, i.e., M could consume a resource N_j in B (with $j \in \{1, \dots, k\}$) and discard the other $k - 1$ resources in a non-confluent manner; this is what we call a *collapsing* approach to non-determinism. On a different direction, Pagani and Ronchi della Rocca [16] proposed λ^r , a resource λ -calculus that implements *non-collapsing* non-determinism, whereby all the possible alternatives for resource consumption are retained together in a sum, ensuring confluence. They investigated typability and characterisations of solvability in λ^r , but no connection with the π -calculus was established. In an attempt to address this gap, our previous work [17] identified λ_{\oplus}^{ζ} , a resource λ -calculus with non-collapsing non-determinism, explicit failure, and *linear* resources (to be used exactly once), and developed a correct typed translation into a session typed π -calculus [5]. The calculus λ_{\oplus}^{ζ} , however, does not include *unrestricted* resources (to be used zero or many times).

This Paper. Here we introduce a new λ -calculus, dubbed $u\lambda_{\oplus}^{\zeta}$, its intersection type system, and its translation into session-typed processes. Our motivation is twofold: to elucidate the status of unrestricted resources in a functional setting with non-collapsing non-determinism, and to characterise unrestricted resources within a translation of functions into processes. Unlike its predecessors, $u\lambda_{\oplus}^{\zeta}$ distinguishes between linear and unrestricted resources. This distinction determines the semantics of terms and especially the deadlocks (*failures*) that arise due to mismatches in resources. This way, $u\lambda_{\oplus}^{\zeta}$ subsumes λ_{\oplus}^{ζ} , which is purely linear and cannot express failures related to unrestricted resources.

Distinguishing linear and unrestricted resources is not a new insight. This idea goes back to Boudol’s λ -calculus with multiplicities [1], where arguments can be tagged as unrestricted. What is new about $u\lambda_{\oplus}^{\zeta}$ is that the distinction between linear and unrestricted resources leads to two main differences. First, occurrences of a variable can be linear or unrestricted, depending on the kind of resources they should be substituted with. This way, e.g., a linear occurrence of variable must be substituted with a linear resource. In $u\lambda_{\oplus}^{\zeta}$, a variable can have linear and unrestricted occurrences in the same term. (Notice that we use the adjective ‘linear’ in connection to resources used exactly once, and not to the number of occurrences of a variable in a term.) Second, failures depend on the nature of the involved resource(s). In $u\lambda_{\oplus}^{\zeta}$, a linear failure arises from a mismatch between required and available (linear) resources; an unrestricted failure arises when a specific (unrestricted) resource is not available.

Accordingly, the syntax of $u\lambda_{\oplus}^{\zeta}$ incorporates linear and unrestricted resources, enabling their consistent separation, within non-collapsing non-determinism. The calculus allows for linear and unrestricted occurrences of variables, as just discussed; bags comprise two separate zones, linear and unrestricted; and the *failure term* $\mathbf{fail}^{x_1, \dots, x_n}$ explicitly mentions the linear variables x_1, \dots, x_n . The (lazy) reduction semantics of $u\lambda_{\oplus}^{\zeta}$ includes two different rules for “fetching” terms from bags, and for consistently handling the failure term.

We equip $u\lambda_{\oplus}^{\zeta}$ with non-idempotent intersection types, extending the approach in [17]: in $u\lambda_{\oplus}^{\zeta}$, intersection types account for more than resource multiplicity, since the elements of the unrestricted bag can have different types. Using intersection types, we define a class of *well-formed* $u\lambda_{\oplus}^{\zeta}$ expressions, which includes terms that correctly consume resources but also terms that may reduce to the failure term. Well-formed expressions thus subsume the *well-typed* expressions that can be defined in a sub-language of $u\lambda_{\oplus}^{\zeta}$ without the failure term.

The calculus $u\lambda_{\oplus}^{\zeta}$ can express terms whose dynamic behaviour is not captured by prior works. This way, e.g., the identity function \mathbf{I} admits two formulations, depending on whether the variable occurrence is linear or unrestricted. One can have $\lambda x.x$, as usual, but also the unrestricted variant $\lambda x.x[i]$, where “[i]” is an index annotation (similar to a qualifier or a tag), which indicates that x should be replaced by the i -th element of the unrestricted zone of the bag. The behaviour of these functions will depend on the bags that are provided as their arguments. Similarly, we can express variants of $\Delta = \lambda x.xx$ and $\Omega = \Delta \Delta$ whose behaviours again depend on linear or unrestricted occurrences of variables and bags. Consider the term $\Delta_7 = \lambda x.(x[1](1 \star \wr x[1] \wr^! \diamond \wr x[2] \wr^!))$, where we use “ \star ” to separate linear and unrestricted resources in the bag, and “ \diamond ” denotes concatenation of unrestricted resources. Term Δ_7 is an abstraction on x of an application of an unrestricted occurrence of x , which aims to consume the first component of an unrestricted bag, to a bag with an empty linear zone (denoted 1) and an unrestricted zone with resources $\wr x[1] \wr^!$ and $\wr x[2] \wr^!$. The self-application $\Delta_7 \Delta_7$ produces a non-terminating behaviour and yet Δ_7 itself is well-formed (see Example 20).

Both $u\lambda_{\oplus}^{\zeta}$ and λ_{\oplus}^{ζ} are *logically motivated* resource λ -calculi, in the following sense: their design has been strongly influenced by $s\pi$, a typed π -calculus resulting from the Curry-Howard correspondence between linear logic and session types in [5], where proofs correspond to processes and cut elimination to process communication. As demonstrated in [5], providing primitive support for explicit failures is key to expressing many useful programming idioms (such as exceptions); this insight is a leading motivation in our design for $u\lambda_{\oplus}^{\zeta}$.

To attest to the logical underpinnings of $u\lambda_{\oplus}^{\zeta}$, we develop a typed translation (or *encoding*) of $u\lambda_{\oplus}^{\zeta}$ into $s\pi$ and establish its correctness with respect to well-established criteria [9, 14]. As in [17], we encode λ_{\oplus}^{ζ} into $s\pi$ by relying on an intermediate language with *sharing* constructs [10, 8, 13]. A key idea in encoding $u\lambda_{\oplus}^{\zeta}$ is to codify the behaviour of unrestricted occurrences of a variable and their corresponding resources in the bag as *client-server connections*, leveraging the copying semantics for the exponential “!A” induced by the Curry-Howard correspondence. This typed encoding into $s\pi$ justifies the semantics of $u\lambda_{\oplus}^{\zeta}$ in terms of precise session protocols (i.e., linear logic propositions, because of the correspondence).

In summary, the **main contributions** of this paper are: (1) The resource calculus $u\lambda_{\oplus}^{\zeta}$ of linear and unrestricted resources, and its associated intersection type system. (2) A typed encoding of $u\lambda_{\oplus}^{\zeta}$ into $s\pi$, which connects well-formed expressions (disciplined by intersection types) and well-typed concurrent processes (disciplined by session types, under the Curry-Howard correspondence with linear logic), subsuming the results in [17].

2 $u\lambda_{\oplus}^{\zeta}$: Unrestricted Resources, Non-Determinism, and Failure

Syntax. We shall use x, y, \dots to range over *variables*, and i, j, \dots , as positive integers, to range over *indices*. Variable occurrences will be *annotated* to distinguish the kind of resource they should be substituted with (linear or unrestricted). With a slight abuse of terminology, we may write “linear variable” and “unrestricted variable” to refer to linear and unrestricted occurrences of a variable. As we will see, a variable’s annotation will be inconsequential for binding purposes. We write \tilde{x} to abbreviate x_1, \dots, x_n , for $n \geq 1$ and each x_i distinct.

► **Definition 1** ($u\widehat{\lambda}_{\oplus}^{\ell}$). We define terms (M, N) , bags (A, B) , and expressions (\mathbb{M}, \mathbb{N}) as:

(Annotations)	$[*] ::= [i] \mid [\ell] \quad i \in \mathbb{N}$
(Terms)	$M, N ::= x[*] \mid \lambda x.M \mid (M B) \mid M\langle\langle B/x \rangle\rangle \mid \mathbf{fail}^{\tilde{x}}$
(Linear Bags)	$C, D ::= 1 \mid \wr M \wr \cdot C$
(Unrestricted Bags)	$U, V ::= 1^! \mid \wr M \wr^! \mid U \diamond V$
(Bags)	$A, B ::= C \star U$
(Expressions)	$\mathbb{M}, \mathbb{N} ::= M \mid \mathbb{M} + \mathbb{N}$

To lighten up notation, we shall omit the annotation for linear variables. This way, e.g., we write $(\lambda x.x)B$ rather than $(\lambda x.x[\ell])B$.

Definition 1 introduces three syntactic categories: *terms* (in functional position); *bags* (multisets of resources, in argument position), and *expressions*, which are finite formal sums that denote possible results of a computation. Below we describe each category in details.

- Terms (unary expressions):
 - Variables: We write $x[\ell]$ to denote a *linear* occurrence of x , i.e., an occurrence that can only be substituted for linear resources. Similarly, $x[i]$ denotes an *unrestricted* occurrence of x , i.e., an occurrence that can only be substituted for a resource located at the i -th position of an unrestricted bag.
 - Abstractions $\lambda x.M$ of a variable x in a term M , which may have contain linear or unrestricted occurrences of x . This way, e.g., $\lambda x.x$ and $\lambda x.x[i]$ are linear and unrestricted versions of the identity function. Notice that the scope of x is M , as usual, and that $\lambda x.(\cdot)$ binds both linear and unrestricted occurrences of x .
 - Applications of a term M to a bag B (written $M B$) and the explicit substitution of a bag B for a variable x (written $\langle\langle B/x \rangle\rangle$) are as expected (cf. [1, 3]). Notice that in $M\langle\langle B/x \rangle\rangle$ the occurrences of x in M , linear and unrestricted, are bound. Some conditions apply to B : this will be evident later on, after we define our operational semantics (cf. Fig. 1).
 - The failure term $\mathbf{fail}^{\tilde{x}}$ denotes a term that will result from a reduction in which there is a lack or excess of resources, where \tilde{x} denotes a multiset of free linear variables that are encapsulated within failure.
- A bag B is defined as $C \star U$: the concatenation of a bag of linear resources C with a bag (actually, a list) of unrestricted resources U . We write $\wr M \wr$ to denote the linear bag that encloses term M , and use $\wr M \wr^!$ in the unrestricted case.
 - Linear bags (C, D, \dots) are multisets of terms. The empty linear bag is denoted 1 . We write $C_1 \cdot C_2$ to denote the concatenation of C_1 and C_2 ; this is a commutative and associative operation, where 1 is the identity.
 - Unrestricted bags (U, V, \dots) are ordered lists of terms. The empty unrestricted bag is denoted as $1^!$. The concatenation of U_1 and U_2 is denoted by $U_1 \diamond U_2$; this operation is associative but not commutative. Given $i \geq 1$, we write U_i to denote the i -th element of the unrestricted (ordered) bag U .
- Expressions are sums of terms, denoted as $\sum_i^n N_i$, where $n > 0$. Sums are associative and commutative; reordering of the terms in a sum is performed silently.

► **Example 2.** Consider the term $M := \lambda x.(x[1] \wr x \wr \star \wr y[1] \wr^!)$, which has linear and unrestricted occurrences of the same variable. This is an abstraction of an application that contains two bound occurrences of x (one unrestricted with index 1, and one linear) and one free unrestricted occurrence of $y[1]$, occurring in an unrestricted bag. As we will see, in $M (C \star U)$, the unrestricted occurrence “ $x[1]$ ” should be replaced by the first element of U .

The salient features of $u\lambda_{\oplus}^{\dagger}$ – the explicit construct for failure, the index annotations on unrestricted variables, the ordering of unrestricted bags – are *design choices* that will be responsible for interesting behaviours, as the following examples illustrate.

► **Example 3.** As already mentioned, $u\lambda_{\oplus}^{\dagger}$ admits different variants of the usual λ -term $\mathbf{I} = \lambda x.x$. We could have one in which x is a linear variable (i.e., $\lambda x.x$), but also several possibilities if x is unrestricted (i.e., $\lambda x.x[i]$, for some positive integer i). Interestingly, because $u\lambda_{\oplus}^{\dagger}$ supports failures, non-determinism, and the consumption of arbitrary terms of the unrestricted bag, these two variants of \mathbf{I} can have behaviours that may differ from the usual interpretation of \mathbf{I} . In Example 11 we will show that the six terms below give different behaviours:

$$\begin{array}{ll} \blacksquare M_1 = (\lambda x.x)(\lambda N \int \star U) & \blacksquare M_4 = (\lambda x.x[1])(1 \star \lambda N \int^! \diamond U) \\ \blacksquare M_2 = (\lambda x.x)(\lambda N_1 \int \cdot \lambda N_2 \int \star U) & \blacksquare M_5 = (\lambda x.x[1])(1 \star 1^! \diamond U) \\ \blacksquare M_3 = (\lambda x.x[1])(\lambda N \int \star 1^!) & \blacksquare M_6 = (\lambda x.x[i])(C \diamond U) \end{array}$$

We will see that M_1, M_4, M_6 reduce without failures, whereas M_2, M_3, M_5 reduce to failure.

► **Example 4.** Similarly, $u\lambda_{\oplus}^{\dagger}$ allows for several forms of the standard λ -terms such as $\Delta := \lambda x.xx$ and $\Omega := \Delta\Delta$, depending on whether the variable x is linear or unrestricted:

1. $\Delta_1 := \lambda x.(x(\lambda x \int \star 1^!))$ consists of an abstraction of a linear occurrence of x applied to a linear bag containing another linear occurrence of x . There are two forms of self-applications of Δ_1 , namely: $\Delta_1(\lambda \Delta_1 \int \star 1^!)$ and $\Delta_1(1 \star \lambda \Delta_1 \int^!)$.
2. $\Delta_4 := \lambda x.(x[1](\lambda x \int \star 1^!))$ consists of an unrestricted occurrence of x applied to a linear bag (containing a linear occurrence of x) that is composed with an empty unrestricted bag. Similarly, there are two self-applications of Δ_4 , namely: $\Delta_4(\lambda \Delta_4 \int \star 1^!)$ and $\Delta_4(1 \star \lambda \Delta_4 \int^!)$.
3. We show applications of an unrestricted variable occurrence ($x[2]$ or $x[1]$) applied to an empty linear bag composed with a non-empty unrestricted bag (of size two):

$\blacksquare \Delta_3 := \lambda x.(x[1](1 \star \lambda x[1] \int^! \diamond \lambda x[1] \int^!))$	$\blacksquare \Delta_6 := \lambda x.(x[1](1 \star \lambda x[1] \int^! \diamond \lambda x[2] \int^!))$
$\blacksquare \Delta_5 := \lambda x.(x[2](1 \star \lambda x[1] \int^! \diamond \lambda x[2] \int^!))$	$\blacksquare \Delta_7 := \lambda x.(x[2](1 \star \lambda x[1] \int^! \diamond \lambda x[1] \int^!))$

Applications between these terms express behaviour, similar to a lazy evaluation of Ω :

$$\begin{array}{ll} \blacksquare \Omega_5 := \Delta_5(1 \star \lambda \Delta_5 \int^! \diamond \lambda \Delta_5 \int^!) & \blacksquare \Omega_{6,5} := \Delta_6(1 \star \lambda \Delta_5 \int^! \diamond \lambda \Delta_6 \int^!) \\ \blacksquare \Omega_{5,6} := \Delta_5(1 \star \lambda \Delta_5 \int^! \diamond \lambda \Delta_6 \int^!) & \blacksquare \Omega_7 := \Delta_7(1 \star \lambda \Delta_7 \int^! \diamond \lambda \Delta_7 \int^!) \end{array}$$

The behaviour of these terms will be made explicit later on (see Examples 13 and 14).

Semantics. The semantics of $u\lambda_{\oplus}^{\dagger}$ captures that linear resources can be used only once, and that unrestricted resources can be used *ad libitum*. Thus, the evaluation of a function applied to a multiset of linear resources produces different possible behaviours, depending on the way these resources are substituted for the linear variables. This induces non-determinism, which we formalise using a *non-collapsing* approach, in which expressions keep all the different possibilities open, and do not commit to one of them. This is in contrast to *collapsing* non-determinism, in which selecting one alternative discards the rest.

We define a reduction relation \longrightarrow , which operates lazily on expressions. Informally, a β -reduction induces an explicit substitution of a bag $B = C \star U$ for a variable x , denoted $\langle\langle B/x \rangle\rangle$, in a term M . This explicit substitution is then expanded depending on whether the head of M has a linear or an unrestricted variable. Accordingly, in $u\lambda_{\oplus}^{\dagger}$ there are *two sources of failure*: one concerns mismatches on linear resources (required vs available resources); the other concerns the unavailability of a required unrestricted resource (an empty bag $1^!$).

To formalise reduction, we require a few auxiliary notions.

► **Definition 5.** *The multiset of free linear variables of \mathbb{M} , denoted $\text{mlfv}(\mathbb{M})$, is defined below. We denote by $[x]$ the multiset containing the linear variable x and $[x_1, \dots, x_n]$ denotes the multiset containing x_1, \dots, x_n . We write $\tilde{x} \uplus \tilde{y}$ to denote the multiset union of \tilde{x} , and \tilde{y} and $\tilde{x} \setminus y$ to express that every occurrence of y is removed from \tilde{x} .*

$$\begin{array}{ll}
 \text{mlfv}(x) = [x] & \text{mlfv}(x[i]) = \text{mlfv}(1) = \emptyset \\
 \text{mlfv}(C \star U) = \text{mlfv}(C) & \text{mlfv}(M B) = \text{mlfv}(M) \uplus \text{mlfv}(B) \\
 \text{mlfv}(\lambda M) = \text{mlfv}(M) & \text{mlfv}(\lambda x.M) = \text{mlfv}(M) \setminus \{x\} \\
 \text{mlfv}(M \langle\langle B/x \rangle\rangle) = (\text{mlfv}(M) \setminus \{x\}) \uplus \text{mlfv}(B) & \text{mlfv}(\lambda M \int \cdot C) = \text{mlfv}(M) \uplus \text{mlfv}(C) \\
 \text{mlfv}(\mathbb{M} + \mathbb{N}) = \text{mlfv}(\mathbb{M}) \uplus \text{mlfv}(\mathbb{N}) & \text{mlfv}(\mathbf{fail}^{x_1, \dots, x_n}) = [x_1, \dots, x_n]
 \end{array}$$

A term M (resp. expression \mathbb{M}) is called linearly closed if $\text{mlfv}(M) = \emptyset$ (resp. $\text{mlfv}(\mathbb{M}) = \emptyset$).

► **Notation 6.** We shall use the following notations.

- $N \in \mathbb{M}$ means that N occurs in the sum \mathbb{M} . Also, we write $N_i \in C$ to denote that N_i occurs in the linear bag C , and $C \setminus N_i$ to denote the linear bag obtained by removing one occurrence of N_i from C .
- $\#(x, M)$ denotes the number of (free) linear occurrences of x in M . Also, $\#(x, \tilde{y})$ denotes the number of occurrences of x in the multiset \tilde{y} .
- $\text{PER}(C)$ is the set of all permutations of a linear bag C and $C_i(n)$ denotes the n -th term in the (permuted) C_i .
- $\text{size}(C)$ denotes the number of terms in a linear bag C . That is, $\text{size}(1) = 0$ and $\text{size}(\lambda M \int \cdot C) = 1 + \text{size}(C)$. Given a bag $B = C \star U$, we define $\text{size}(B)$ as $\text{size}(C)$.

► **Definition 7 (Head).** Given a term M , we define $\text{head}(M)$ inductively as:

$$\begin{array}{lll}
 \text{head}(x) = x & \text{head}(M B) = \text{head}(M) & \text{head}(\lambda x.M) = \lambda x.M \\
 \text{head}(x[i]) = x[i] & \text{head}(\mathbf{fail}^{\tilde{x}}) = \mathbf{fail}^{\tilde{x}} & \text{head}(M \langle\langle B/x \rangle\rangle) = \begin{cases} \text{head}(M) & \text{if } \#(x, M) = \text{size}(B) \\ \mathbf{fail}^\emptyset & \text{otherwise} \end{cases}
 \end{array}$$

► **Definition 8 (Head Substitution).** Let M be a term such that $\text{head}(M) = x$. The head substitution of a term N for x in M , denoted $M\{N/x\}$, is inductively defined as follows (where $x \neq y$):

$$x\{N/x\} = N \quad (M B)\{N/x\} = (M\{N/x\}) B \quad (M \langle\langle B/y \rangle\rangle)\{N/x\} = (M\{N/x\}) \langle\langle B/y \rangle\rangle$$

When $\text{head}(M) = x[i]$, the head substitution $M\{N/x[i]\}$ works as expected: $x[i]\{N/x[i]\} = N$ as the base case of the definition. Finally, we define contexts for terms and expressions:

► **Definition 9 (Evaluation Contexts).** Contexts for terms ($C\text{Term}$) and expressions ($C\text{Expr}$) are defined by the following grammar:

$$(C\text{Term}) \quad C[\cdot], C'[\cdot] ::= ([\cdot])B \mid ([\cdot])\langle\langle B/x \rangle\rangle \quad (C\text{Expr}) \quad D[\cdot], D'[\cdot] ::= M + [\cdot]$$

Reduction is defined by the rules in Fig. 1. Rule $[\mathbf{R} : \text{Beta}]$ induces explicit substitutions. Resource consumption is implemented by two fetch rules, which open up explicit substitutions:

- Rule $[\mathbf{R} : \text{Fetch}^\ell]$, the *linear fetch*, ensures that the number of required resources matches the size of the linear bag C . It induces a sum of terms with head substitutions, each denoting the partial evaluation of an element from C . Thus, the size of C determines the summands in the resulting expression.
- Rule $[\mathbf{R} : \text{Fetch}^!]$, the *unrestricted fetch*, consumes a resource occurring in a specific position of the unrestricted bag U via a linear head substitution of an unrestricted variable occurring in the head of the term. In this case, reduction results in an explicit substitution with U kept unaltered. Note that we check for the size of the linear bag C : in the case $\#(x, M) \neq \text{size}(C)$, the term evolves to a linear failure via Rule $[\mathbf{R} : \text{fail}^\ell]$ (see Example 12). This is another design choice: linear failure is prioritised in $u\lambda_{\oplus}^{\ell}$.

$$\begin{array}{c}
\text{[R : Beta]} \frac{}{(\lambda x.M)B \longrightarrow M\langle\langle B/x \rangle\rangle} \\
\text{[R : Fetch}^\ell] \frac{\text{head}(M) = x \quad C = \{N_1\} \cdots \{N_k\}, k \geq 1 \quad \#(x, M) = k}{M\langle\langle C \star U/x \rangle\rangle \longrightarrow M\{N_1/x\}\langle\langle (C \setminus N_1) \star U/x \rangle\rangle + \cdots + M\{N_k/x\}\langle\langle (C \setminus N_k) \star U/x \rangle\rangle} \\
\text{[R : Fetch}^!] \frac{\text{head}(M) = x[i] \quad \#(x, M) = \text{size}(C) \quad U_i = \{N\}^!}{M\langle\langle C \star U/x \rangle\rangle \longrightarrow M\{N/x[i]\}\langle\langle C \star U/x \rangle\rangle} \\
\text{[R : Fail}^\ell] \frac{\#(x, M) \neq \text{size}(C) \quad \tilde{y} = (\text{mlfv}(M) \setminus x) \uplus \text{mlfv}(C)}{M\langle\langle C \star U/x \rangle\rangle \longrightarrow \sum_{\text{PER}(C)} \text{fail}^{\tilde{y}}} \\
\text{[R : Fail}^!] \frac{\#(x, M) = \text{size}(C) \quad U_i = \mathbf{1}^! \quad \text{head}(M) = x[i]}{M\langle\langle C \star U/x \rangle\rangle \longrightarrow M\{\text{fail}^0/x[i]\}\langle\langle C \star U/x \rangle\rangle} \\
\text{[R : Cons}_1] \frac{\tilde{y} = \text{mlfv}(C)}{(\text{fail}^x) C \star U \longrightarrow \sum_{\text{PER}(C)} \text{fail}^{x \uplus \tilde{y}}} \\
\text{[R : Cons}_2] \frac{\#(z, \tilde{x}) = \text{size}(C) \quad \tilde{y} = \text{mlfv}(C)}{\text{fail}^x \langle\langle C \star U/z \rangle\rangle \longrightarrow \sum_{\text{PER}(C)} \text{fail}^{(x \setminus z) \uplus \tilde{y}}} \\
\text{[R : ECont]} \frac{\mathbb{M} \longrightarrow \mathbb{M}'}{D[\mathbb{M}] \longrightarrow D[\mathbb{M}']} \quad \text{[R : TCont]} \frac{M \longrightarrow \sum_{i=1}^k M'_i}{C[M] \longrightarrow \sum_{i=1}^k C[M'_i]}
\end{array}$$

■ **Figure 1** Reduction rules for $u\lambda_{\oplus}^{\ddagger}$.

Four rules show reduction to failure terms, and accumulate free variables involved in failed reductions. Rules $\text{[R : Fail}^\ell]$ and $\text{[R : Fail}^!]$ formalise the failure to evaluate an explicit substitution $M\langle\langle C \star U/x \rangle\rangle$. The former rule targets a linear failure, which occurs when the size of C does not match the number of occurrences of x . The multiset \tilde{y} preserves all free linear variables in M and C . The latter rule targets an *unrestricted failure*, which occurs when the head of the term is $x[i]$ and U_i (i.e., the i -th element of U) is empty. In this case, failure preserves the free linear variables in M and C excluding the head unrestricted occurrence $x[i]$ which is replaced by fail^0 .

Rules $\text{[R : Cons}_1]$ and $\text{[R : Cons}_2]$ describe reductions that lazily consume the failure term, when a term has fail^x at its head position. The former rule consumes bags attached to it whilst preserving all its free linear variables; the latter rule consumes explicit substitution attached to it whilst also preserving all its free linear variables. The side condition $\#(z, \tilde{x}) = \text{size}(C)$ is necessary in Rule $\text{[R : Cons}_2]$ to avoid a clash with the premise of Rule $\text{[R : Fail}^\ell]$. Finally, Rules [R : ECont] and [R : TCont] state closure by the C and D contexts (cf. Def. 9).

Notice that the left-hand sides of the reduction rules in $u\lambda_{\oplus}^{\ddagger}$ do not interfere with each other. As a result, reduction in $u\lambda_{\oplus}^{\ddagger}$ satisfies a *diamond property*: for all $\mathbb{M} \in u\lambda_{\oplus}^{\ddagger}$, if there exist $\mathbb{M}_1, \mathbb{M}_2 \in u\lambda_{\oplus}^{\ddagger}$ such that $\mathbb{M} \longrightarrow \mathbb{M}_1$ and $\mathbb{M} \longrightarrow \mathbb{M}_2$, then there exists $\mathbb{N} \in u\lambda_{\oplus}^{\ddagger}$ such that $\mathbb{M}_1 \longrightarrow \mathbb{N} \longleftarrow \mathbb{M}_2$ (see [18] for more details).

► **Notation 10.** As usual, \longrightarrow^* denotes the reflexive-transitive closure of \longrightarrow . We write $\mathbb{N} \longrightarrow_{\text{[R]}} \mathbb{M}$ to denote that [R] is the last (non-contextual) rule used in the step from \mathbb{N} to \mathbb{M} .

► **Example 11** (Cont. Example 3). We illustrate different reductions for $\lambda x.x$ and $\lambda x.x[i]$.

1. $M_1 = (\lambda x.x)(\lambda N \int \star U)$ concerns a linear variable x with an linear bag containing one element. This is similar to the usual meaning of applying an identity function to a term:
 $(\lambda x.x)(\lambda N \int \star U) \xrightarrow{[\mathbf{R}:\mathbf{Beta}]} x \langle \langle \lambda N \int \star U / x \rangle \rangle \xrightarrow{[\mathbf{R}:\mathbf{Fetch}^\ell]} x \{ \{ N/x \} \} \langle \langle 1 \star U / x \rangle \rangle = N \langle \langle 1 \star U / x \rangle \rangle$,
 with a “garbage collector” that collects unused unrestricted resources.

2. $M_2 = (\lambda x.x)(\lambda N_1 \int \cdot \lambda N_2 \int \star U)$ concerns the case in which a linear variable x has a single occurrence but the linear bag has size two. Term M_2 reduces to a sum of failure terms:
 $(\lambda x.x)(\lambda N_1 \int \cdot \lambda N_2 \int \star U) \xrightarrow{[\mathbf{R}:\mathbf{Beta}]} x \langle \langle \lambda N_1 \int \cdot \lambda N_2 \int \star U / x \rangle \rangle \xrightarrow{[\mathbf{R}:\mathbf{Fail}^\ell]} \sum_{\text{PER}(C)} \mathbf{fail}^{\tilde{y}}$

for $C = \lambda N_1 \int \cdot \lambda N_2 \int$ and $\tilde{y} = \text{mlfv}(C)$.

3. $M_3 = (\lambda x.x[1])(\lambda N \int \star 1^!)$ represents an abstraction of an unrestricted variable, which aims to consume the first element of the unrestricted bag. Because this bag is empty, M_3 reduces to failure:

$$(\lambda x.x[1])(\lambda N \int \star 1^!) \xrightarrow{[\mathbf{R}:\mathbf{Beta}]} x[1] \langle \langle \lambda N \int \star 1^! / x \rangle \rangle \xrightarrow{[\mathbf{R}:\mathbf{fail}^\ell]} \mathbf{fail}^{\tilde{y}},$$

for $\tilde{y} = \text{mlfv}(N)$. Notice that $0 = \#(x, x[1]) \neq \text{size}(\lambda N \int) = 1$, since there are no linear occurrences of x in $x[1]$.

► **Example 12.** To illustrate the need to check “size(C)” in $[\mathbf{R}:\mathbf{Fail}^!]$, consider the term $x[1] \langle \langle \lambda M \int \star 1^! / x \rangle \rangle$, which features both a mismatch of linear bags for the linear variables to be substituted and an empty unrestricted bag with the need for the first element to be substituted. We check the size of the linear bag because we wish to prioritise the reduction of Rule $[\mathbf{R}:\mathbf{Fail}^\ell]$. Hence, in case of a mismatch of linear resources we wish not to perform a reduction via Rule $[\mathbf{R}:\mathbf{Fail}^!]$. This is a design choice: our semantics collapses linear failure at the earliest moment it arises.

► **Example 13** (Cont. Example 4). Self-applications of Δ_1 do not behave as an expected variation of a lazy reduction from Ω . Both $\Delta_1(\lambda \Delta_1 \int \star 1^!)$ and $\Delta_1(1 \star \lambda \Delta_1 \int^!)$ reduce to failure since the number of linear occurrences of x does not match the number of resources in the linear bag: $\Delta_1(\lambda \Delta_1 \int \star 1^!) \xrightarrow{(\lambda x.(x \int \star 1))} \langle \langle \lambda \Delta_1 \int \star 1^! / x \rangle \rangle \xrightarrow{} \mathbf{fail}^\emptyset$.

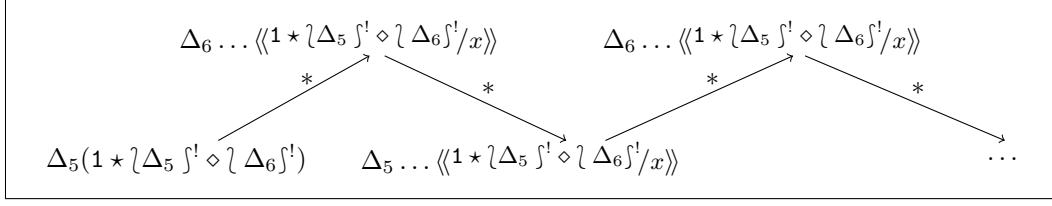
The term $\Delta_4(1 \star \lambda \Delta_4 \int^!)$ also fails: the linear bag is empty and there is one linear occurrence of x in Δ_4 . Note that $\Delta_4(\lambda \Delta_4 \int \star \lambda \Delta_4 \int^!)$ reduces to another application of Δ_4 before failing:

$$\begin{aligned} \Delta_4(\lambda \Delta_4 \int \star \lambda \Delta_4 \int^!) &= (\lambda x.(x[1](\lambda x \int \star 1^!)))(\lambda \Delta_4 \int \star \lambda \Delta_4 \int^!) \\ &\xrightarrow{[\mathbf{R}:\mathbf{Beta}]} (x[1](\lambda x \int \star 1^!)) \langle \langle \lambda \Delta_4 \int \star \lambda \Delta_4 \int^! / x \rangle \rangle \\ &\xrightarrow{[\mathbf{R}:\mathbf{Fetch}^!]} (\Delta_4(\lambda x \int \star 1^!)) \langle \langle \lambda \Delta_4 \int \star \lambda \Delta_4 \int^! / x \rangle \rangle \\ &\xrightarrow{*} \mathbf{fail}^\emptyset \langle \langle \lambda x \int \star 1^! / y \rangle \rangle \langle \langle \lambda \Delta_4 \int \star \lambda \Delta_4 \int^! / x \rangle \rangle \end{aligned}$$

Differently from [17], there are terms in $u\lambda_{\oplus}^{\zeta}$ that when applied to each other behave similarly to Ω , namely $\Omega_{5,6}$, $\Omega_{6,5}$, and Ω_7 (Example 4).

► **Example 14** (Cont. Example 4). The following reductions illustrate different behaviours provided that subtle changes are made within $u\lambda_{\oplus}^{\zeta}$ -terms:

- An interesting behaviour of $u\lambda_{\oplus}^{\zeta}$ is that variations of Δ can be applied to each other and appear alternately (highlighted in blue) in the functional position throughout the computation – this behaviour is illustrated in Fig. 2:



■ **Figure 2** An Ω -like behaviour in $u\lambda_{\oplus}^{\zeta}$ (cf. Example 14).

$$\begin{aligned}
\Omega_{5,6} &= \Delta_5(1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!) \\
&= (\lambda x.(x[2](1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!))) (1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!) \\
&\rightarrow_{[\text{R:Beta}]} (x[2](1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!)) \langle\langle 1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (\Delta_6(1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!)) \langle\langle 1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Beta}]} (y[1](1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!)) \langle\langle (1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!) / y \rangle\rangle \langle\langle 1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (x[1](1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!)) \langle\langle (1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!) / y \rangle\rangle \langle\langle 1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (\Delta_5(1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!)) \langle\langle (1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^!) / y \rangle\rangle \langle\langle 1 \star \Delta_5 \mathcal{J}^! \diamond \Delta_6 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow \dots
\end{aligned}$$

- Applications of Δ_7 into two unrestricted copies of Δ_7 behave as Ω producing a non-terminating behaviour. Letting $B = 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!$, we have:

$$\begin{aligned}
\Omega_7 &= (\lambda x.(x[2](1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!))) (1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!) \\
&\rightarrow_{[\text{R:Beta}]} (x[2](1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!)) \langle\langle 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (\Delta_7(1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!)) \langle\langle 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Beta}]} (y[2](1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!)) \langle\langle B / y \rangle\rangle \langle\langle 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (x[1](1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!)) \langle\langle B / y \rangle\rangle \langle\langle 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow_{[\text{R:Fetch}^!]} (\Delta_7(1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^!)) \langle\langle B / y \rangle\rangle \langle\langle 1 \star \Delta_7 \mathcal{J}^! \diamond \Delta_7 \mathcal{J}^! / x \rangle\rangle \\
&\rightarrow \dots
\end{aligned}$$

Later on we will show that this term is well-formed (see Example 20) with respect to the intersection type system introduced in § 3.

3 Well-Formed Expressions via Intersection Types

We define *well-formed* $u\lambda_{\oplus}^{\zeta}$ -expressions by relying on a non-idempotent intersection type system, based on the system by Bucciarelli et al. [4]. Our system for well-formed expressions subsumes the one in [17]: it uses *strict* and *multiset* types to check linear bags; moreover, it uses *list* and *tuple* types to check unrestricted bags. As in [17], we write “well-formedness” (of terms, bags, and expressions) to stress that, unlike usual type systems, our system can account for terms that may reduce to the failure term (cf. Remark 22).

- **Definition 15** (Types for $u\lambda_{\oplus}^{\zeta}$). We define *strict*, *multiset*, *list*, and *tuple* types.

$$\begin{array}{ll}
(\textit{Strict}) & \sigma, \tau, \delta ::= \mathbf{unit} \mid (\pi, \eta) \rightarrow \sigma \\
(\textit{Multiset}) & \pi, \zeta ::= \bigwedge_{i \in I} \sigma_i \mid \omega \\
(\textit{List}) & \eta, \epsilon ::= \sigma \mid \epsilon \diamond \eta \\
(\textit{Tuple}) & (\pi, \eta)
\end{array}$$

11:10 Unrestricted Resources in Encoding Functions as Processes

A strict type can be the **unit** type or a functional type $(\pi, \eta) \rightarrow \sigma$, where (π, η) is a tuple type and σ is a strict type. Multiset types can be either the empty type ω or an intersection of strict types $\bigwedge_{i \in I} \sigma_i$, with I non-empty. The operator \wedge is commutative, associative, non-idempotent, that is, $\sigma \wedge \sigma \neq \sigma$, with identity ω . The intersection type $\bigwedge_{i \in I} \sigma_i$ is the type of a linear bag; the cardinality of I corresponds to its size.

A list type can be either a strict type σ or the composition $\epsilon \diamond \eta$ of two list types ϵ and η . We use the list type $\epsilon \diamond \eta$ to type the concatenation of two unrestricted bags. A tuple type (π, η) types the concatenation of a linear bag of type π with an unrestricted bag of type η . Notice that a list type $\epsilon \diamond \eta$ can be recursively unfolded into a finite composition of strict types $\sigma_1 \diamond \dots \diamond \sigma_n$, for some $n \geq 1$. In this case the length of $\epsilon \diamond \eta$ is n and that σ_i is its i -th strict type, for $1 \leq i \leq n$.

► **Notation 16.** Given $k \geq 0$, we write σ^k to stand for $\sigma \wedge \dots \wedge \sigma$ (k times, if $k > 0$) or for ω (if $k = 0$). Similarly, $\hat{x} : \sigma^k$ stands for $x : \sigma, \dots, x : \sigma$ (k times, if $k > 0$) or for $x : \omega$ (if $k = 0$). Given $k \geq 1$, we write $x^1 : \eta$ to stand for $x[1] : \eta_1, \dots, x[k] : \eta_k$.

► **Notation 17** ($\eta \propto \epsilon$). Let ϵ and η be two list types, with the length of ϵ greater or equal to that of η . Let us write ϵ_i and η_i to denote the i -th strict type in ϵ and η , respectively. We write $\eta \propto \epsilon$ meaning the initial sublist, whenever there exist ϵ' and ϵ'' such that: i) $\epsilon = \epsilon' \diamond \epsilon''$; ii) the size of ϵ' is that of η ; iii) for all i , $\epsilon'_i = \eta_i$.

Linear contexts range over Γ, Δ, \dots and unrestricted contexts range over Θ, Υ, \dots . They are defined by the following grammar:

$$\Gamma, \Delta ::= - \mid x : \sigma \mid \Gamma, x : \sigma \quad \Theta, \Upsilon ::= - \mid x^1 : \eta \mid \Theta, x^1 : \eta$$

The empty linear/unrestricted type assignment is denoted “-”. Linear variables can occur more than once in a linear context; they are assigned only strict types. For instance, $x : (\tau, \sigma) \rightarrow \tau, x : \tau$ is a valid context: it means that x can be of both type $(\tau, \sigma) \rightarrow \tau$ and τ . In contrast, unrestricted variables can occur at most once in unrestricted contexts; they are assigned only list types. The multiset of linear variables in Γ is denoted as $\text{dom}(\Gamma)$; similarly, $\text{dom}(\Theta)$ denotes the set of unrestricted variables in Θ .

Judgements are of the form $\Theta; \Gamma \models \mathbb{M} : \sigma$, where the left-hand side contexts are separated by “;” and $\mathbb{M} : \sigma$ means that \mathbb{M} has type σ . We write $\models \mathbb{M} : \sigma$ to denote $-; - \models \mathbb{M} : \sigma$.

► **Definition 18** (Well-formed $u\lambda_{\oplus}^{\downarrow}$ expressions). An expression \mathbb{M} is well-formed (wf, for short) if there exist Γ, Θ and τ such that $\Theta; \Gamma \models \mathbb{M} : \tau$ is entailed via the rules in Fig. 3.

We describe the well-formedness rules in Fig. 3.

- Rules $[\mathbf{F} : \text{var}^{\ell}]$ and $[\mathbf{F} : \text{var}^1]$ assign types to linear and unrestricted variables, respectively.
- Rule $[\mathbf{F} : \text{var}^1]$ resembles the *copy* rule [6] where we use a linear copy of an unrestricted variable $x[i]$ of type σ , typed with $x^1 : \eta$, and type the linear copy with the corresponding strict type η_i which in this case the linear copy x would have type equal to σ .
- Rules $[\mathbf{F} : 1^{\ell}]$ and $[\mathbf{F} : 1^1]$ assign types to the empty linear/unrestricted bag: 1 has type ω , whereas 1^1 has an arbitrary strict type σ . Arbitrariness is allowed since the substitution of an unrestricted variable for 1^1 leads to a **fail** term (Rule $[\mathbf{R} : \text{Fail}^1]$), which has an arbitrary strict type.
- Rule $[\mathbf{F} : \text{abs}]$ assigns type $(\sigma^k, \eta) \rightarrow \tau$ to an abstraction $\lambda z.M$, provided that the unrestricted occurrences of z may be typed by the unrestricted context containing $z^1 : \eta$, the linear occurrences of z are typed with the linear context containing $\hat{z} : \sigma^k$, for some $k \geq 0$, and there are no other linear occurrences of z in the linear context Γ .

$\text{[F:var}^\ell] \frac{}{\Theta; x : \sigma \models x : \sigma}$	$\text{[F:var}^!] \frac{\Theta, x^! : \eta; x : \eta_i, \Delta \models x : \sigma}{\Theta, x^! : \eta; \Delta \models x[i] : \sigma}$	$\text{[F:1}^\ell] \frac{}{\Theta; - \models 1 : \omega}$
$\text{[F:1}^!] \frac{}{\Theta; - \models 1^! : \sigma}$	$\text{[F:abs]} \frac{\Theta, z^! : \eta; \Gamma, \hat{z} : \sigma^k \models M : \tau \quad z \notin \text{dom}(\Gamma)}{\Theta; \Gamma \models \lambda z.M : (\sigma^k, \eta) \rightarrow \tau}$	
$\text{[F:app]} \frac{\Theta; \Gamma \models M : (\sigma^j, \eta) \rightarrow \tau \quad \Theta; \Delta \models B : (\sigma^k, \epsilon) \quad \eta \propto \epsilon}{\Theta; \Gamma, \Delta \models M B : \tau}$		$\text{[F:ex-sub]} \frac{\Theta, x^! : \eta; \Gamma, \hat{x} : \sigma^j \models M : \tau \quad \Theta; \Delta \models B : (\sigma^k, \epsilon) \quad \eta \propto \epsilon}{\Theta; \Gamma, \Delta \models M \langle\langle B/x \rangle\rangle : \tau}$
$\text{[F:bag]} \frac{\Theta; \Gamma \models C : \sigma^k \quad \Theta; - \models U : \eta}{\Theta; \Gamma \models C \star U : (\sigma^k, \eta)}$	$\text{[F:bag}^\ell] \frac{\Theta; \Gamma \models M : \sigma \quad \Theta; \Delta \models C : \sigma^k}{\Theta; \Gamma, \Delta \models \{M\} \cdot C : \sigma^{k+1}}$	
$\text{[F:bag}^!] \frac{\Theta; - \models M : \sigma}{\Theta; - \models \{M\}^! : \sigma}$	$\text{[F:} \diamond \text{bag}^!] \frac{\Theta; - \models U : \epsilon \quad \Theta; - \models V : \eta}{\Theta; - \models U \diamond V : \epsilon \diamond \eta}$	$\text{[F:fail]} \frac{\text{dom}(\Gamma) = \tilde{x}}{\Theta; \Gamma \models \text{fail}^{\tilde{x}} : \tau}$
$\text{[F:sum]} \frac{\Theta; \Gamma \models M : \sigma \quad \Theta; \Gamma \models N : \sigma}{\Theta; \Gamma \models M + N : \sigma}$		$\text{[F:weak]} \frac{\Theta; \Gamma \models M : \sigma \quad x \notin \text{dom}(\Gamma)}{\Theta; \Gamma, x : \omega \models M : \sigma}$

■ **Figure 3** Well-formedness rules for $u\lambda_{\oplus}^{\ddagger}$ (cf. Def. 18). In Rules [F:app] and [F:ex-sub]: $k, j \geq 0$.

- Rules [F:app] and [F:ex-sub] (for application and explicit substitution, resp.) use the condition $\eta \propto \epsilon$ (cf. Notation 17), which captures the portion of the unrestricted bag that is effectively used in a term: it ensures that ϵ can be decomposed into some ϵ' and ϵ'' , such that each type component ϵ'_i matches with η_i . If this requirement is satisfied, Rule [F:app] types an application $M B$ given that M has a functional type in which the left of the arrow is a tuple type (σ^j, η) whereas the bag B is typed with tuple (σ^k, ϵ) . Similarly, Rule [F:ex-sub] types the term $M \langle\langle B/x \rangle\rangle$ provided that B has the tuple type (σ^k, ϵ) and M is typed with the variable x having linear type assignment σ^j and unrestricted type assignment η .

► **Remark 19.** Differently from intersection type systems [4, 16], in Rules [F:app] and [F:ex-sub] there is no equality requirement between j and k , as we would like to capture terms that fail due to a mismatch in resources: we only require that the linear part of the tuples are composed of the same strict type, say σ . As a term can take an unrestricted bag with arbitrary size we only require that the elements of the unrestricted bag that are used have a “consistent” type, i.e., the type of the unrestricted bag satisfies the relation \propto with the unrestricted fragment of the corresponding tuple type.

There are four rules for bags:

- Rule [F:bag[!]] types an unrestricted bag $\{M\}^!$ with the type σ of M . Note that $\{x\}^!$, an unrestricted bag containing a linear variable x , is not well-formed, whereas $\{x[i]\}^!$ is well-formed.
- Rule [F:bag] assigns the tuple type (σ^k, η) to the concatenation of a linear bag of type σ^k with an unrestricted bag of type η .
- Rules [F:bag^ℓ] and [F:◊bag[!]] type the concatenation of linear and unrestricted bags.
- Rule [F:1[!]] allows an empty unrestricted bag to have an arbitrary σ type since it may be referred to by a variable for substitution: we must be able to compare its type with the type of unrestricted variables that may consume the empty bag (this reduction would inevitably lead to failure).

11:12 Unrestricted Resources in Encoding Functions as Processes

As in [17], Rule [F:fail] handles the failure term, and is the main difference with respect to standard type systems. Rules for sums and weakening ([F : sum] and [F : weak]) are standard.

► **Example 20** (Cont. Example 14). Term $\Delta_7 := \lambda x.x[2](1 \star \lambda x[1] \mathfrak{f}^! \diamond \lambda x[1] \mathfrak{f}^!)$ is well-formed, as ensured by the judgement $\Theta; - \models \Delta_7 : (\omega, \sigma' \diamond (\sigma^j, \sigma' \diamond \sigma') \rightarrow \tau) \rightarrow \tau$, whose derivation is given below:

- Π_3 is the derivation of $\Theta, x^! : \eta; - \models \lambda x[1] \mathfrak{f}^! : \sigma'$, for $\eta = \sigma' \diamond (\sigma^j, \sigma' \diamond \sigma') \rightarrow \tau$.
- Π_4 is the derivation: $\Theta, x^! : \eta; - \models x[2] : (\sigma^j, \sigma' \diamond \sigma') \rightarrow \tau$
- Π_5 is the derivation: $\Theta, x^! : \eta; x : \omega \models (1 \star \lambda x[1] \mathfrak{f}^! \diamond \lambda x[1] \mathfrak{f}^!) : (\omega, \sigma' \diamond \sigma')$

Therefore,

$$\frac{\frac{\text{[F:app]}}{\text{[F:abs]}} \frac{\frac{\Pi_5 \quad \Pi_4 \quad \sigma' \diamond \sigma' \propto \sigma' \diamond \sigma'}{\Theta, x^! : \eta; x : \omega \models x[2](1 \star \lambda x[1] \mathfrak{f}^! \diamond \lambda x[1] \mathfrak{f}^!) : \tau}}{\Theta; - \models \underbrace{\lambda x.(x[2](1 \star \lambda x[1] \mathfrak{f}^! \diamond \lambda x[1] \mathfrak{f}^!))}_{\Delta_7} : (\omega, \eta) \rightarrow \tau}}{\Theta; - \models \lambda x.(x[2](1 \star \lambda x[1] \mathfrak{f}^! \diamond \lambda x[1] \mathfrak{f}^!)) : (\omega, \eta) \rightarrow \tau}}$$

Well-formed expressions satisfy subject reduction (SR); see [18] for a full proof.

► **Theorem 21** (SR in $u\lambda_{\oplus}^!$). *If $\Theta; \Gamma \models \mathbb{M} : \tau$ and $\mathbb{M} \rightarrow \mathbb{M}'$ then $\Theta; \Gamma \models \mathbb{M}' : \tau$.*

Proof. By structural induction on the reduction rules. We proceed by analysing the rule applied in \mathbb{M} . An interesting case occurs when the rule is [F : Fetch[!]]: Then $\mathbb{M} = M \langle\langle C \star U/x \rangle\rangle$, where $U = \lambda N_1 \mathfrak{f}^! \diamond \dots \diamond \lambda N_l \mathfrak{f}^!$ and $\text{head}(M) = x[i]$. The reduction is as follows:

$$\text{[R : Fetch}^! \text{]} \frac{\text{head}(M) = x[i] \quad U_i = \lambda N_i \mathfrak{f}^!}{M \langle\langle C \star U/x \rangle\rangle \rightarrow M \{\{N_i/x[i]\}\} \langle\langle C \star U/x \rangle\rangle}$$

By hypothesis, one has the derivation:

$$\text{[F:ex-sub]} \frac{\Theta, x^! : \eta; \Gamma', \hat{x} : \sigma^j \models M : \tau \quad \frac{\text{[F:bag]} \frac{\frac{\Pi}{\Theta; \cdot \models U : \epsilon} \quad \Theta; \Delta \models C : \sigma^k}}{\Theta; \Delta \models C \star U : (\sigma^k, \epsilon)} \quad \eta \propto \epsilon}}{\Theta; \Gamma', \Delta \models M \langle\langle C \star U/x \rangle\rangle : \tau}$$

where Π has the form

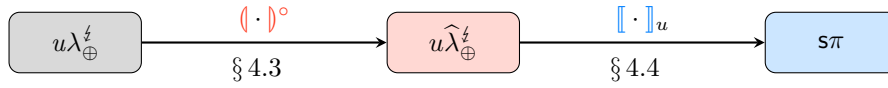
$$\text{[F:} \diamond \text{bag}^! \text{]} \frac{\frac{\text{[F:bag}^! \text{]} \frac{\Theta; \cdot \models N_1 : \epsilon_1}{\Theta; \cdot \models \lambda N_1 \mathfrak{f}^! : \epsilon_1} \quad \dots \quad \text{[F:bag}^! \text{]} \frac{\Theta; \cdot \models N_l : \epsilon_l}{\Theta; \cdot \models \lambda N_l \mathfrak{f}^! : \epsilon_l}}{\Theta; \cdot \models \lambda N_1 \mathfrak{f}^! \diamond \dots \diamond \lambda N_l \mathfrak{f}^! : \epsilon}}$$

with $\Gamma = \Gamma', \Delta$. Notice that if $\epsilon_i = \delta$ and $\eta \propto \epsilon$ then $\eta_i = \delta$. It can be shown that there exists a derivation Π_1 of $\Theta, x^! : \eta; \Gamma', \hat{x} : \sigma^j \models M \{\{N_i/x[i]\}\} : \tau$. Therefore, we have:

$$\text{[F:ex-sub]} \frac{\Theta, x^! : \eta; \Gamma', \hat{x} : \sigma^j \models M \{\{N_i/x[i]\}\} : \tau \quad \frac{\text{[F:bag]} \frac{\Theta; \cdot \models U : \epsilon \quad \Theta; \Delta \models C : \sigma^k}}{\Theta; \Delta \models C \star U : (\sigma^k, \epsilon)} \quad \eta \propto \epsilon}}{\Theta; \Gamma', \Delta \models M \{\{N_i/x[i]\}\} \langle\langle C \star U/x \rangle\rangle : \tau}$$

◀

► **Remark 22** (Well-Formed vs Well-Typed Expressions). Our type system (and Theorem 21) can be specialised to the case of *well-typed* expressions that do not contain (and never reduce to) the failure term. In particular, Rules [F:app] and [F:ex-sub] would need to check that $\sigma^k = \sigma^j$, as failure can be caused due to a mismatch of linear resources. A difference between well typed and well formed expressions is that the former satisfy subject expansion, but the latter do not: expressions that lead to failure can be ill-typed yet failure itself is well-formed.



■ **Figure 4** Our two-step approach to encode $u\lambda_{\oplus}^{\zeta}$ into $s\pi$.

4 A Typed Encoding of $u\lambda_{\oplus}^{\zeta}$ into Concurrent Processes

We encode $u\lambda_{\oplus}^{\zeta}$ into $s\pi$, a session π -calculus that stands on a Curry-Howard correspondence between linear logic and session types (§ 4.1). We extend the two-step approach that we devised in [17] for the sub-calculus λ_{\oplus}^{ζ} (with linear resources only) (cf. Fig. 4). First, in § 4.3, we define an encoding $(\cdot)^{\circ}$ from well-formed expressions in $u\lambda_{\oplus}^{\zeta}$ to well-formed expressions in a variant of $u\lambda_{\oplus}^{\zeta}$ with *sharing*, dubbed $u\widehat{\lambda}_{\oplus}^{\zeta}$ (§ 4.2). Then, in § 4.4, we define an encoding $[\cdot]_u$ (for a name u) from well-formed expressions in $u\widehat{\lambda}_{\oplus}^{\zeta}$ to well-typed processes in $s\pi$.

We prove that $(\cdot)^{\circ}$ and $[\cdot]_u$ satisfy well-established correctness criteria [9, 14]: *type preservation*, *operational completeness*, *operational soundness*, and *success sensitiveness* (cf. [18]). Because $u\lambda_{\oplus}^{\zeta}$ includes unrestricted resources, the results given here strictly generalise those in [17].

4.1 $s\pi$: A Session-Typed π -Calculus

$s\pi$ is a π -calculus with *session types* [11, 12], which ensure that the endpoints of a channel perform matching actions. We consider the full process framework in [5], including constructs for specifying labelled choices and client/server connections; they will be useful to codify unrestricted resources and variables in $u\lambda_{\oplus}^{\zeta}$. Following [6, 19], $s\pi$ stands on a Curry-Howard correspondence between session types and a linear logic with dual modalities/types ($\&A$ and $\oplus A$), which define *non-deterministic* session behaviour. As in [6, 19], in $s\pi$, cut elimination corresponds to communication, proofs to processes, and propositions to session types.

Syntax. Names $x, y, z, w \dots$ denote the endpoints of protocols specified by session types. We write $P\{x/y\}$ for the capture-avoiding substitution of x for y in process P .

► **Definition 23** (Processes). *The syntax of $s\pi$ processes is given by the grammar below.*

$$\begin{aligned}
 P, Q ::= & \mathbf{0} \mid \bar{x}(y).P \mid x(y).P \mid x.l_i; P \mid x.\text{case}_{i \in I}\{l_i : P_i\} \mid x.\overline{\text{close}} \mid x.\text{close}; P \\
 & \mid (P \mid Q) \mid [x \leftrightarrow y] \mid (\nu x)P \mid !x(y).P \mid \bar{x}?(y).P \\
 & \mid x.\overline{\text{some}}; P \mid x.\overline{\text{none}} \mid x.\text{some}_{(w_1, \dots, w_n)}; P \mid (P \oplus Q)
 \end{aligned}$$

Process $\mathbf{0}$ denotes inaction. Process $\bar{x}(y).P$ sends a fresh name y along x and then continues as P . Process $x(y).P$ receives a name z along x and then continues as $P\{z/y\}$. Process $x.\text{case}_{i \in I}\{l_i : P_i\}$ is a branching construct, with labelled alternatives indexed by the finite set I : it awaits a choice on x with continuation P_j for each $j \in I$. Process $x.l_i; P$ selects on x the alternative indexed by i before continuing as P . Processes $x.\overline{\text{close}}$ and $x.\text{close}; P$ are complementary actions for closing session x . We sometimes use the shorthand notations $\bar{y}[]$ and $y[]; P$ to stand for $y.\overline{\text{close}}$ and $y.\text{close}; P$, respectively. Process $P \mid Q$ is the parallel execution of P and Q . The forwarder process $[x \leftrightarrow y]$ denotes a bi-directional link between sessions x and y . Process $(\nu x)P$ denotes the process P in which name x is kept private (local) to P . Process $!x(y).P$ defines a server that spawns copies of P upon requests on x . Process $\bar{x}?(y).P$ denotes a client that connects to a server by sending the fresh name y on x .

$\bar{x}(y).Q \mid x(y).P \longrightarrow (\nu y)(Q \mid P)$	$x.\overline{\text{some}}; P \mid x.\text{some}_{(w_1, \dots, w_n)}; Q \longrightarrow P \mid Q$
$Q \longrightarrow Q' \Rightarrow P \oplus Q \longrightarrow P \oplus Q'$	$x.\overline{\text{close}} \mid x.\text{close}; P \longrightarrow P$
$x.1_i; Q \mid x.\text{case}_{i \in I} \{1_i : P_i\} \longrightarrow Q \mid P_i$	$!x(y).Q \mid \bar{x}(y).P \longrightarrow (\nu x)(!x(y).Q \mid (\nu y)(Q \mid P))$
$(\nu x)([x \leftrightarrow y] \mid P) \longrightarrow P\{y/x\} \quad (x \neq y)$	$P \equiv P' \wedge P' \longrightarrow Q' \wedge Q' \equiv Q \Rightarrow P \longrightarrow Q$
$Q \longrightarrow Q' \Rightarrow P \mid Q \longrightarrow P \mid Q'$	$P \longrightarrow Q \Rightarrow (\nu y)P \longrightarrow (\nu y)Q$
$x.\overline{\text{none}} \mid x.\text{some}_{(w_1, \dots, w_n)}; Q \longrightarrow w_1.\overline{\text{none}} \mid \dots \mid w_n.\overline{\text{none}}$	

■ **Figure 5** Reduction for $\mathfrak{s}\pi$.

The remaining constructs come from [5] and introduce non-deterministic sessions which *may* provide a session protocol *or* fail. Process $x.\overline{\text{some}}; P$ confirms that the session on x will execute and continues as P . Process $x.\overline{\text{none}}$ signals the failure of implementing the session on x . Process $x.\text{some}_{(w_1, \dots, w_n)}; P$ specifies a dependency on a non-deterministic session x . This process can either (i) synchronise with an action $x.\overline{\text{some}}$ and continue as P , or (ii) synchronise with an action $x.\overline{\text{none}}$, discard P , and propagate the failure on x to (w_1, \dots, w_n) , which are sessions implemented in P . When x is the only session implemented in P , there is no tuple of dependencies (w_1, \dots, w_n) and so we write simply $x.\text{some}; P$. Finally, process $P \oplus Q$ denotes a non-deterministic choice between P and Q . We shall often write $\bigoplus_{i \in \{1, \dots, n\}} P_i$ to stand for $P_1 \oplus \dots \oplus P_n$. In $(\nu y)P$ and $x(y).P$ the occurrence of name y is binding, with scope P . The set of free names of P is denoted by $fn(P)$.

Semantics. The *reduction relation* of $\mathfrak{s}\pi$ specifies the computations that a process performs on its own (cf. Fig. 5). It is closed by *structural congruence*, denoted \equiv , which expresses basic identities for processes and the non-collapsing nature of non-determinism (cf. [18]).

The first reduction rule formalises communication, which concerns bound names only (internal mobility), as y is bound in $\bar{x}(y).Q$ and $x(y).P$. Reduction for the forwarder process leads to a substitution. The reduction rule for closing a session is self-explanatory, as is the rule in which prefix $x.\overline{\text{some}}$ confirms the availability of a non-deterministic session. When a non-deterministic session is not available, $x.\overline{\text{none}}$ triggers this failure to all dependent sessions w_1, \dots, w_n ; this may in turn trigger further failures (i.e., on sessions that depend on w_1, \dots, w_n). The remaining rules define contextual reduction with respect to restriction, composition, and non-deterministic choice.

Type System. Session types govern the behaviour of the names of a process. An assignment $x : A$ enforces the use of name x according to the protocol specified by A .

► **Definition 24** (Session Types). *Session types are given by*

$$A, B ::= \perp \mid \mathbf{1} \mid A \otimes B \mid A \wp B \mid \bigoplus_{i \in I} \{1_i : A_i\} \mid \&_{i \in I} \{1_i : A_i\} \mid !A \mid ?A \mid \&A \mid \oplus A$$

The multiplicative units \perp and $\mathbf{1}$ are used to type closed session endpoints. We use $A \otimes B$ to type a name that first outputs a name of type A before proceeding as specified by B . Similarly, $A \wp B$ types a name that first inputs a name of type A before proceeding as specified by B . Then, $!A$ types a name that repeatedly provides a service specified by A . Dually, $?A$ is the type of a name that can connect to a server offering A . Types $\bigoplus_{i \in I} \{1_i : A_i\}$ and $\&_{i \in I} \{1_i : A_i\}$ are assigned to names that can select and offer a labelled choice, respectively. Then we have the two modalities introduced in [5]. We use $\&A$ as the type of a (non-deterministic) session that *may produce* a behaviour of type A . Dually, $\oplus A$ denotes the type of a session that *may consume* a behaviour of type A .

$\text{[Tid]} \frac{}{[x \leftrightarrow y] \vdash x:A, y:\bar{A}; \Theta}$	$\text{[T1]} \frac{}{x.\text{close} \vdash x : \mathbf{1}; \Theta}$	$\text{[T}\perp\text{]} \frac{P \vdash \Delta; \Theta}{x.\text{close}; P \vdash x:\perp, \Delta; \Theta}$
$\text{[T}\otimes\text{]} \frac{P \vdash \Delta, y : A; \Theta \quad Q \vdash \Delta', x : B; \Theta}{\bar{x}(y).(P \mid Q) \vdash \Delta, \Delta', x : A \otimes B; \Theta}$	$\text{[T}\wp\text{]} \frac{P \vdash \Delta, y : C, x : D; \Theta}{x(y).P \vdash \Delta, x : C \wp D; \Theta}$	
$\text{[T}\oplus_{\tilde{w}}^x\text{]} \frac{P \vdash \tilde{w} : \&\Delta, x : A; \Theta}{x.\text{some}_{\tilde{w}}; P \vdash \tilde{w}:\&\Delta, x : \oplus A; \Theta}$	$\text{[T}\&_{\tilde{d}}^x\text{]} \frac{P \vdash \Delta, x : A; \Theta}{x.\overline{\text{some}}; P \vdash \Delta, x : \&A; \Theta}$	
$\text{[T}\&^x\text{]} \frac{}{x.\overline{\text{none}} \vdash x : \&A; \Theta}$	$\text{[T}\oplus\text{]} \frac{P \vdash \&\Delta; \Theta \quad Q \vdash \&\Delta; \Theta}{P \oplus Q \vdash \&\Delta; \Theta}$	
$\text{[T}\oplus_i\text{]} \frac{P \vdash \Delta, x : A_i; \Theta}{x.\mathbf{1}_i; P \vdash \Delta, x : \oplus_{i \in I} \{\mathbf{1}_i : A_i\}; \Theta}$	$\text{[T}\&\text{]} \frac{P_i \vdash \Delta, x : A_i; \Theta \quad (\forall i \in I)}{x.\text{case}_{i \in I} \{\mathbf{1}_i : P_i\} \vdash \Delta, x : \&_{i \in I} \{\mathbf{1}_i : A_i\}; \Theta}$	
$\text{[T?]} \frac{P \vdash \Delta; x : A, \Theta}{P \vdash \Delta, x : ?A; \Theta}$	$\text{[T!]} \frac{P \vdash y : A; \Theta}{!x(y).P \vdash x : !A; \Theta}$	$\text{[Tcopy]} \frac{P \vdash \Delta, y : A; x : A, \Theta}{\bar{x}?(y).P \vdash \Delta; x : A, \Theta}$

■ **Figure 6** Typing rules for $\mathfrak{s}\pi$.

The two endpoints of a session should be *dual* to ensure absence of communication errors. The dual of a type A is denoted \bar{A} . Duality corresponds to negation $(\cdot)^\perp$ in linear logic [5].

► **Definition 25** (Duality). *Duality on types is given by:*

$$\begin{array}{llllll} \bar{\perp} = \perp & \bar{\mathbf{1}} = \mathbf{1} & \overline{A \otimes B} = \bar{A} \wp \bar{B} & \overline{\oplus_{i \in I} \{\mathbf{1}_i : A_i\}} = \&_{i \in I} \{\mathbf{1}_i : \bar{A}_i\} & \overline{\oplus A} = \&\bar{A} \\ \overline{!A} = ?A & \overline{?A} = !A & \overline{A \wp B} = \bar{A} \otimes \bar{B} & \overline{\&_{i \in I} \{\mathbf{1}_i : A_i\}} = \oplus_{i \in I} \{\mathbf{1}_i : \bar{A}_i\} & \overline{\&A} = \oplus \bar{A} \end{array}$$

Judgements are of the form $P \vdash \Delta; \Theta$, where P is a process, Δ is the linear context, and Θ is the unrestricted context. Both Δ and Θ contain assignments of types to names, but satisfy different substructural principles: while Θ satisfies weakening, contraction and exchange, Δ only satisfies exchange. The empty context is denoted “.”. We write $\&\Delta$ to denote that all assignments in Δ have a non-deterministic type, i.e., $\Delta = w_1:\&A_1, \dots, w_n:\&A_n$, for some A_1, \dots, A_n . The typing judgement $P \vdash \Delta$ corresponds to the logical sequent for classical linear logic, which can be recovered by erasing processes and name assignments.

Typing rules for processes in Fig. 6 correspond to proof rules in linear logic; we discuss some of them. Rule [Tid] interprets the identity axiom using the forwarder process. Rules [T1] and [T \perp] type the process constructs for session termination. Rules [T \otimes] and [T \wp] type output and input of a name along a session, resp. The last four rules are used to type process constructs related to non-determinism and failure. Rules [T $\oplus_{\tilde{w}}^x$] and [T $\&^x$] introduce a session of type $\&A$, which may produce a behaviour of type A : while the former rule covers the case in which $x : A$ is indeed available, the latter rule formalises the case in which $x : A$ is not available (i.e., a failure). Given a sequence of names $\tilde{w} = w_1, \dots, w_n$, Rule [T $\oplus_{\tilde{w}}^x$] accounts for the possibility of not being able to consume the session $x : A$ by considering sessions different from x as potentially not available. Rule [T \oplus] expresses non-deterministic choice of processes P and Q that implement non-deterministic behaviours only. Finally, Rule [T \oplus_i] and [T $\&$] correspond, resp., to selection and branching: the former provides a selection of behaviours along x as long as P is guarded with the i -th behaviour; the latter offers a labelled choice where each behaviour A_i is matched to a corresponding P_i .

The type system enjoys type preservation, a result that follows from the cut elimination property in linear logic; it ensures that the observable interface of a system is invariant under reduction. The type system also ensures other properties for well-typed processes (e.g. global progress, strong normalisation, and confluence); see [5] for details.

► **Theorem 26** (Type Preservation [5]). *If $P \vdash \Delta; \Theta$ and $P \longrightarrow Q$ then $Q \vdash \Delta; \Theta$.*

4.2 $u\widehat{\lambda}_{\oplus}^{\dagger}$: An Auxiliary Calculus With Sharing

To facilitate the encoding of $u\lambda_{\oplus}^{\dagger}$ into $\mathfrak{s}\pi$, we define $u\widehat{\lambda}_{\oplus}^{\dagger}$: an auxiliary calculus whose constructs are inspired by the work of Gundersen et al. [10], Ghilezan et al. [8], and Kesner and Lengrand [13]. The syntax of $u\widehat{\lambda}_{\oplus}^{\dagger}$ only modifies the syntax of terms, which is defined by the grammar below; variables $x[*]$, bags B , and expressions \mathbb{M} are as in Definition 1.

$$\begin{aligned} \text{(Terms)} \quad M, N, L ::= & x[*] \mid \lambda x.(M[\tilde{x} \leftarrow x]) \mid (M B) \mid M \langle N/x \rangle \mid M \llbracket U/x \rrbracket \\ & \mid \mathbf{fail}^{\tilde{x}} \mid M[\tilde{x} \leftarrow x] \mid (M[\tilde{x} \leftarrow x]) \langle\langle B/x \rangle\rangle \end{aligned}$$

We consider the *sharing construct* $M[\tilde{x} \leftarrow x]$ and two kinds of explicit substitutions: the *explicit linear substitution*, written $M \langle N/x \rangle$, and the *explicit unrestricted substitution*, written $M \llbracket U/x \rrbracket$. The term $M[\tilde{x} \leftarrow x]$ defines the sharing of variables \tilde{x} occurring in M using the linear variable x . We shall refer to x as *sharing variable* and to \tilde{x} as *shared variables*. A linear variable is only allowed to appear once in a term. Notice that \tilde{x} can be empty: $M[\leftarrow x]$ expresses that x does not share any variables in M . As in $u\lambda_{\oplus}^{\dagger}$, the term $\mathbf{fail}^{\tilde{x}}$ explicitly accounts for failed attempts at substituting the variables in \tilde{x} .

We summarise some requirements. In $M[\tilde{x} \leftarrow x]$, we require: (i) every $x_i \in \tilde{x}$ occurs exactly once in M and that (ii) x_i is not a sharing variable. The occurrence of x_i can appear within the fail term $\mathbf{fail}^{\tilde{y}}$, if $x_i \in \tilde{y}$. In the explicit linear substitution $M \langle N/x \rangle$, we require: the variable x has to occur in M ; x cannot be a sharing variable; and x cannot be in an explicit linear substitution occurring in M ; all free *linear* occurrences of x in M are bound. In the explicit unrestricted substitution $M \llbracket U/x \rrbracket$, we require: all free *unrestricted* occurrences of x in M are bound; x cannot be in an explicit unrestricted substitution occurring in M . This way, e.g., $M' \langle L/x \rangle \langle N/x \rangle$ and $M' \langle U'/x \rangle \llbracket U/x \rrbracket$ are not valid terms in $u\widehat{\lambda}_{\oplus}^{\dagger}$.

The following congruence will be important when proving encoding correctness.

► **Definition 27.** *The congruence \equiv_{λ} for $u\widehat{\lambda}_{\oplus}^{\dagger}$ on terms and expressions is given by the identities below.*

$$\begin{aligned} M \llbracket U/x \rrbracket & \equiv_{\lambda} M, x \notin M \\ (MB) \langle N/x \rangle & \equiv_{\lambda} (M \langle N/x \rangle) B, x \notin \mathbf{fv}(B) \\ (MB) \llbracket U/x \rrbracket & \equiv_{\lambda} (M \llbracket U/x \rrbracket) B, x \notin \mathbf{fv}(B) \\ (MA)[\tilde{x} \leftarrow x] \langle\langle B/x \rangle\rangle & \equiv_{\lambda} (M[\tilde{x} \leftarrow x] \langle\langle B/x \rangle\rangle) A, x_i \in \tilde{x} \Rightarrow x_i \notin \mathbf{fv}(A) \\ M[\tilde{y} \leftarrow y] \langle\langle A/y \rangle\rangle [\tilde{x} \leftarrow x] \langle\langle B/x \rangle\rangle & \equiv_{\lambda} (M[\tilde{x} \leftarrow x] \langle\langle B/x \rangle\rangle) [\tilde{y} \leftarrow y] \langle\langle A/y \rangle\rangle, \\ & \quad x_i \in \tilde{x} \Rightarrow x_i \notin \mathbf{fv}(A), y_i \in \tilde{y} \Rightarrow y_i \notin \mathbf{fv}(B) \\ M \langle N_2/y \rangle \langle N_1/x \rangle & \equiv_{\lambda} M \langle N_1/x \rangle \langle N_2/y \rangle, x \notin \mathbf{fv}(N_2), y \notin \mathbf{fv}(N_1) \\ M \llbracket U_2/y \rrbracket \llbracket U_1/x \rrbracket & \equiv_{\lambda} M \llbracket U_1/x \rrbracket \llbracket U_2/y \rrbracket, x \notin \mathbf{fv}(U_2), y \notin \mathbf{fv}(U_1) \\ C[M] & \equiv_{\lambda} C[M'], \text{ with } M \equiv_{\lambda} M' \\ D[\mathbb{M}] & \equiv_{\lambda} D[\mathbb{M}'], \text{ with } \mathbb{M} \equiv_{\lambda} \mathbb{M}' \end{aligned}$$

The first rule states that we may remove unneeded unrestricted substitutions when the variable in concern does not appear within the term. The next three identities enforce that bags can always be moved in and out of all forms of explicit substitution, which are useful to manipulate expressions and to form a redex for Rule [R : Beta]. The other rules deal with permutation of explicit substitutions and contextual closure.

Well-formedness for $u\widehat{\lambda}_{\oplus}^{\dagger}$, based on intersection types, is defined as in §3; see [18].

$$\begin{array}{l}
(x)^\bullet = x \qquad (x[i])^\bullet = x[i] \qquad (1)^\bullet = 1 \\
(1^!)^\bullet = 1^! \qquad (\mathbf{fail}^x)^\bullet = \mathbf{fail}^x \qquad (M B)^\bullet = (M)^\bullet (B)^\bullet \\
(\lambda M \int^!)^\bullet = \lambda M \int^! \qquad (\lambda M \int \cdot C)^\bullet = \lambda (M)^\bullet \int \cdot (C)^\bullet \qquad (C \star U)^\bullet = (C)^\bullet \star (U)^\bullet \\
(U \diamond V)^\bullet = U \diamond V \qquad (M \langle N/x \rangle)^\bullet = (M)^\bullet \langle (N)^\bullet / x \rangle \qquad (M \ll U/x \gg)^\bullet = (M)^\bullet \ll (U)^\bullet / x \gg \\
(\lambda x.M)^\bullet = \lambda x.((M \langle x_1, \dots, x_n/x \rangle)^\bullet [x_1, \dots, x_n \leftarrow x]) \quad \#(x, M) = n, \text{ each } x_i \text{ is fresh} \\
(M \langle C \star U/x \rangle)^\bullet = \begin{cases} \sum_{C_i \in \text{PER}(C)^\bullet} ((M \langle \tilde{x}/x \rangle)^\bullet \langle C_i(1)/x_1 \rangle \cdots \langle C_i(k)/x_k \rangle \ll U/x \gg), & \text{if } \#(x, M) = \text{size}(C) = k \\ (M \langle x_1, \dots, x_k/x \rangle)^\bullet [x_1, \dots, x_k \leftarrow x] \ll (C \star U)^\bullet / x \gg, & \text{if } \#(x, M) = k \geq 0 \end{cases}
\end{array}$$

■ **Figure 7** Auxiliary Encoding: $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ into $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$.

4.3 Encoding $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ into $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$

We define an encoding $(\cdot)^\circ$ from well-formed terms in $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ into $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$. This encoding relies on an intermediate encoding $(\cdot)^\bullet$ on $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ -terms.

► **Notation 28.** Given a term M such that $\#(x, M) = k$ and a sequence of pairwise distinct fresh variables $\tilde{x} = x_1, \dots, x_k$ we write $M \langle \tilde{x}/x \rangle$ or $M \langle x_1, \dots, x_k/x \rangle$ to stand for $M \langle x_1/x \rangle \cdots \langle x_k/x \rangle$, i.e., a simultaneous linear substitution whereby each distinct linear occurrence of x in M is replaced by a distinct $x_i \in \tilde{x}$. Notice that each x_i has the same type as x . We use (simultaneous) linear substitutions to force all bound linear variables in $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ to become shared variables in $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$.

► **Definition 29** (From $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ to $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$). Let $M \in u\lambda_{\oplus}^{\frac{k}{\oplus}}$. Suppose $\Theta; \Gamma \models M : \tau$, with $\text{dom}(\Gamma) = \text{lfv}(M) = \{x_1, \dots, x_k\}$ and $\#(x_i, M) = j_i$. We define $(M)^\circ$ as

$$(M)^\circ = (M \langle \tilde{x}_1/x_1 \rangle \cdots \langle \tilde{x}_k/x_k \rangle)^\bullet [\tilde{x}_1 \leftarrow x_1] \cdots [\tilde{x}_k \leftarrow x_k]$$

where $\tilde{x}_i = x_{i_1}, \dots, x_{i_{j_i}}$ and the encoding $(\cdot)^\bullet : u\lambda_{\oplus}^{\frac{k}{\oplus}} \rightarrow u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$ is defined in Fig. 7 on $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ -terms. The encoding $(\cdot)^\circ$ extends homomorphically to expressions.

The encoding $(\cdot)^\circ$ converts n occurrences of x in a term into n distinct variables x_1, \dots, x_n . The sharing construct coordinates them by constraining each to occur exactly once within a term. We proceed in two stages. First, we share all linear free linear variables using $(\cdot)^\circ$: this ensures that free variables are replaced by shared variables which are then bound by the sharing construct. Second, we apply the encoding $(\cdot)^\bullet$ on the corresponding term. The encoding is presented in Fig. 7: $(\cdot)^\bullet$ maintains $x[i]$ unaltered, and acts homomorphically over concatenation of bags and explicit substitutions. The encoding renames bound variables with bound shared variables. As we will see, this will enable a tight operational correspondence result with $s\pi$. In [18] we establish the correctness of $(\cdot)^\circ$.

► **Example 30.** We apply the encoding $(\cdot)^\bullet$ in some of the $u\lambda_{\oplus}^{\frac{k}{\oplus}}$ -terms from Example 3: for simplicity, we assume that N and U have no free variables.

$$\begin{aligned}
((\lambda x.x) \lambda N \int \star U)^\bullet &= ((\lambda x.x)^\bullet)^\bullet (\lambda N \int \star U)^\bullet = \lambda x.x_1[x_1 \leftarrow x] \lambda (N)^\bullet \int \star (U)^\bullet \\
((\lambda x.x[1]) \mathbf{1} \star \lambda N \int^! \diamond U)^\bullet &= ((\lambda x.x[1])^\bullet)^\bullet (\mathbf{1} \star \lambda N \int^! \diamond U)^\bullet = (\lambda x.x[1][\leftarrow x]) \mathbf{1} \star \lambda (N)^\bullet \int^! \diamond (U)^\bullet
\end{aligned}$$

4.4 Encoding $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$ into $s\pi$

We now define our encoding of $u\widehat{\lambda}_{\oplus}^{\frac{k}{\oplus}}$ into $s\pi$, and establish its correctness.

► **Notation 31.** To help illustrate the behaviour of the encoding, we use the names x , x^ℓ , and $x^!$ to denote three distinct channel names: while x^ℓ is the channel that performs the linear substitution behaviour of the encoded term, channel $x^!$ performs the unrestricted behaviour.

► **Definition 32** (From $u\widehat{\lambda}_{\oplus}^{\downarrow}$ into $s\pi$: Expressions). Let u be a name. The encoding $\llbracket \cdot \rrbracket_u : u\widehat{\lambda}_{\oplus}^{\downarrow} \rightarrow s\pi$ is defined in Fig. 8.

Every (free) variable x in an $u\widehat{\lambda}_{\oplus}^{\downarrow}$ expression becomes a name x in its corresponding $s\pi$ process. As customary in encodings of λ into π , we use a name u to provide the behaviour of the encoded expression. In our case, u is a non-deterministic session: the encoded expression can be effectively available or not; this is signalled by prefixes $u.\overline{\text{some}}$ and $u.\overline{\text{none}}$, respectively.

We discuss the most salient aspects of the encoding in Fig. 8.

- While linear variables are encoded as in [17], the encoding of an unrestricted variable $x[j]$, not treated in [17], is much more interesting: it first connects to a server along channel x via a request $x^!(x_i)$ followed by a selection on $x_i.l_j$, which takes the j -th branch.
- The encoding of $\lambda x.M[\tilde{x} \leftarrow x]$ confirms its behaviour first followed by the receiving of a channel x . The channel x provides a linear channel x^ℓ and an unrestricted channel $x^!$ for dedicated substitutions of the linear and unrestricted bag components.
- We encode $M(C \star U)$ as a non-deterministic sum: an application involves a choice in the order in which the elements of C are substituted.
- The encoding of $C \star U$ synchronises with the encoding of $\lambda x.M[\tilde{x} \leftarrow x]$. The channel x^ℓ provides the linear behaviour of the bag C while $x^!$ provides the behaviour of U ; this is done by guarding the encoding of U with a server connection such that every time a channel synchronises with $!x^!(x_i)$ a fresh copy of U is spawned.
- The encoding of $\wr M \wr \cdot C$ synchronises with the encoding of $M[\tilde{x} \leftarrow x]$, just discussed. The name y_i is used to trigger a failure in the computation if there is a lack of elements in the encoding of the bag.
- The encoding of $M[\tilde{x} \leftarrow x]$ first confirms the availability of the linear behaviour along x^ℓ . Then it sends a name y_i , which is used to collapse the process in the case of a failed reduction. Subsequently, for each shared variable, the encoding receives a name, which will act as an occurrence of the shared variable. At the end, a failure prefix on x is used to signal that there is no further information to send over.
- The encoding of U synchronises with the last half encoding of $x[j]$; the name x_i selects the j -th term in the unrestricted bag.
- The encoding of $M \langle N/x \rangle$ is the composition of the encodings of M and N , where we await a confirmation of a behaviour along the variable that is being substituted.
- $M \llbracket U/x \rrbracket$ is encoded as the composition of the encoding of M and a server guarding the encoding of U : in order for $\llbracket M \rrbracket_u$ to gain access to $\llbracket U \rrbracket_{x_i}$ it must first synchronise with the server channel $x^!$ to spawn a fresh copy of U .
- The encoding of $\mathbb{M} + \mathbb{N}$ homomorphically preserves non-determinism. Finally, the encoding of $\text{fail}^{x_1, \dots, x_k}$ simply triggers failure on u and on each of x_1, \dots, x_k .

► **Example 33.** [Cont. Example 3] We illustrate the encoding $\llbracket \cdot \rrbracket$ on the $u\widehat{\lambda}_{\oplus}^{\downarrow}$ -terms/bags occurring in $M_1 = \lambda x.x_1[x_1 \leftarrow x](\wr \langle N \rangle \wr \wr \star \langle U \rangle \star)$ as below:

$$\begin{aligned} \llbracket \lambda x.x_1[x_1 \leftarrow x] \rrbracket_v &= v.\overline{\text{some}}; v(x).x.\overline{\text{some}}; x(x^\ell).x(x^!).x[]; \llbracket x_1[x_1 \leftarrow x] \rrbracket_v \\ \llbracket \wr \langle N \rangle \wr \wr \star \langle U \rangle \star \rrbracket_x &= x.\text{some}_{\text{fv}(\wr \langle N \rangle \wr \wr \star)}; \overline{x}(x^\ell).(\llbracket \langle N \rangle \rrbracket_{x^\ell} \mid \overline{x}(x^!).(!x^!(x_i).\llbracket \langle U \rangle \rrbracket_{x_i} \mid \overline{x}[])) \end{aligned}$$

$$\begin{aligned}
\llbracket x \rrbracket_u &= x.\overline{\text{some}}; [x \leftrightarrow u] \\
\llbracket x[j] \rrbracket_u &= \overline{x^1?}(x_i).x_i.l_j; [x_i \leftrightarrow u] \\
\llbracket \lambda x.M[\tilde{x} \leftarrow x] \rrbracket_u &= u.\overline{\text{some}}; u(x).x.\overline{\text{some}}; x(x^\ell).x(x^1).x.\overline{\text{close}}; \llbracket M[\tilde{x} \leftarrow x] \rrbracket_u \\
\llbracket M[\tilde{x} \leftarrow x] \llbracket C \star U/x \rrbracket \rrbracket_u &= \bigoplus_{C_i \in \text{PER}(C)} (\nu x)(x.\overline{\text{some}}; x(x^\ell).x(x^1).x.\overline{\text{close}}; \llbracket M[\tilde{x} \leftarrow x] \rrbracket_u \mid \llbracket C_i \star U \rrbracket_x) \\
\llbracket M(C \star U) \rrbracket_u &= \bigoplus_{C_i \in \text{PER}(C)} (\nu v)(\llbracket M \rrbracket_v \mid v.\overline{\text{some}}_{u, \text{fv}(C)}; \overline{v}(x).([v \leftrightarrow u] \mid \llbracket C_i \star U \rrbracket_x)) \\
\llbracket C \star U \rrbracket_x &= x.\overline{\text{some}}_{\text{fv}(C)}; \overline{x}(x^\ell).(\llbracket C \rrbracket_{x^\ell} \mid \overline{x}(x^1).(!x^1(x_i).\llbracket U \rrbracket_{x_i} \mid x.\overline{\text{close}})) \\
\llbracket \llbracket M \rrbracket \cdot C \rrbracket_{x^\ell} &= x^\ell.\overline{\text{some}}_{\text{fv}(\llbracket M \rrbracket \cdot C)}; x^\ell(y_i).x^\ell.\overline{\text{some}}_{y_i, \text{fv}(\llbracket M \rrbracket \cdot C)}; x^\ell.\overline{\text{some}}; \overline{x^\ell}(x_i). \\
&\quad (x_i.\overline{\text{some}}_{\text{fv}(M)}; \llbracket M \rrbracket_{x_i} \mid \llbracket C \rrbracket_{x^\ell} \mid y_i.\overline{\text{none}}) \\
\llbracket \mathbf{1} \rrbracket_{x^\ell} &= x^\ell.\overline{\text{some}}_\emptyset; x^\ell(y_n).(y_n.\overline{\text{some}}; y_n.\overline{\text{close}} \mid x^\ell.\overline{\text{some}}_\emptyset; x^\ell.\overline{\text{none}}) \\
\llbracket \mathbf{1}^! \rrbracket_x &= x.\overline{\text{none}} \\
\llbracket \llbracket N \rrbracket^! \rrbracket_x &= \llbracket N \rrbracket_x \\
\llbracket U \rrbracket_x &= x.\text{case}_{U_i \in U} \{ \mathbf{1}_i : \llbracket U_i \rrbracket_x \} \\
\llbracket M \llbracket N/x \rrbracket \rrbracket_u &= (\nu x)(\llbracket M \rrbracket_u \mid x.\overline{\text{some}}_{\text{fv}(N)}; \llbracket N \rrbracket_x) \\
\llbracket M \llbracket U/x \rrbracket \rrbracket_u &= (\nu x^1)(\llbracket M \rrbracket_u \mid !x^1(x_i).\llbracket U \rrbracket_{x_i}) \\
\llbracket M[\leftarrow x] \rrbracket_u &= x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_i).(y_i.\overline{\text{some}}_{u, \text{fv}(M)}; y_i.\overline{\text{close}}; \llbracket M \rrbracket_u \mid x^\ell.\overline{\text{none}}) \\
\llbracket M[x_1, \dots, x_n \leftarrow x] \rrbracket_u &= x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\overline{\text{some}}_\emptyset; y_1.\overline{\text{close}}; \mathbf{0} \mid \\
&\quad x^\ell.\overline{\text{some}}; x^\ell.\overline{\text{some}}_{u, (\text{fv}(M) \setminus \{x_1, \dots, x_n\})}; x^\ell(x_1).\llbracket M[x_2, \dots, x_n \leftarrow x] \rrbracket_u) \\
\llbracket M \mid N \rrbracket_u &= \llbracket M \rrbracket_u \oplus \llbracket N \rrbracket_u \\
\llbracket \text{fail}^{x_1, \dots, x_k} \rrbracket_u &= u.\overline{\text{none}} \mid x_1.\overline{\text{none}} \mid \dots \mid x_k.\overline{\text{none}}
\end{aligned}$$

■ **Figure 8** Encoding $u\widehat{\lambda}_{\oplus}^{\xi}$ into $s\pi$ (cf. Def. 32).

$$\begin{aligned}
\llbracket \llbracket M_1 \rrbracket^\bullet \rrbracket_u &= \llbracket \lambda x.x_1[x_1 \leftarrow x] \llbracket \llbracket N \rrbracket^\bullet \rrbracket \star \llbracket \llbracket U \rrbracket^\bullet \rrbracket \rrbracket_u \\
&= (\nu v)(\llbracket \lambda x.x_1[x_1 \leftarrow x] \rrbracket_v \mid v.\overline{\text{some}}_{u, \text{fv}(\llbracket N \rrbracket^\bullet)}; \overline{v}(x).([v \leftrightarrow u] \mid \llbracket \llbracket \llbracket N \rrbracket^\bullet \rrbracket \star \llbracket \llbracket U \rrbracket^\bullet \rrbracket \rrbracket_x)) \\
&= (\nu v)(v.\overline{\text{some}}; v(x).x.\overline{\text{some}}; x(x^\ell).x(x^1).x[]; x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\overline{\text{some}}_\emptyset; y_1[]; \mathbf{0} \mid \\
&\quad x^\ell.\overline{\text{some}}; x^\ell.\overline{\text{some}}_u; x^\ell(x_1).x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2).(y_2.\overline{\text{some}}_{u, x_1}; y_2[]; \llbracket x_1 \rrbracket_v \mid x^\ell.\overline{\text{none}})) \mid \\
&\quad v.\overline{\text{some}}_{u, \text{fv}(\llbracket N \rrbracket^\bullet)}; \overline{v}(x).([v \leftrightarrow u] \mid \\
&\quad x.\overline{\text{some}}_{\text{fv}(\llbracket N \rrbracket^\bullet)}; \overline{x}(x^\ell).(x^\ell.\overline{\text{some}}_{\text{fv}(\llbracket N \rrbracket^\bullet)}; x^\ell.\overline{\text{some}}_{y_1, \text{fv}(\llbracket N \rrbracket^\bullet \rrbracket)}; \\
&\quad x^\ell.\overline{\text{some}}; \overline{x^\ell}(x_1).(x_1.\overline{\text{some}}_{\text{fv}(\llbracket N \rrbracket^\bullet)}; \llbracket \llbracket N \rrbracket^\bullet \rrbracket_{x_1} \mid y_1.\overline{\text{none}} \mid x^\ell.\overline{\text{some}}_\emptyset; x^\ell(y_2). \\
&\quad (y_2.\overline{\text{some}}; \overline{y_2}[] \mid x^\ell.\overline{\text{some}}_\emptyset; x^\ell.\overline{\text{none}})) \mid \overline{x}(x^1).(!x^1(x_i).\llbracket \llbracket U \rrbracket^\bullet \rrbracket_{x_i} \mid \overline{x}[])))
\end{aligned}$$

We now encode intersection types (for $u\widehat{\lambda}_{\oplus}^{\xi}$) into session types (for $s\pi$).

► **Definition 34** (From $u\widehat{\lambda}_{\oplus}^{\xi}$ into $s\pi$: Types). *The translation $\llbracket \cdot \rrbracket$ in Figure 9 extends as follows to a context $\Gamma = x_1:\sigma_1, \dots, x_m:\sigma_m, v_1:\pi_1, \dots, v_n:\pi_n$ and a context $\Theta = x_1^!:\eta_1, \dots, x_k^!:\eta_k$:*

$$\begin{aligned}
\llbracket \Gamma \rrbracket &= x_1 : \&[\overline{\sigma_1}], \dots, x_m : \&[\overline{\sigma_m}], v_1 : \overline{\llbracket \pi_1 \rrbracket}_{(\sigma, i_1)}, \dots, v_n : \overline{\llbracket \pi_n \rrbracket}_{(\sigma, i_n)} \\
\llbracket \Theta \rrbracket &= x_1^! : \overline{\llbracket \eta_1 \rrbracket}, \dots, x_k^! : \overline{\llbracket \eta_k \rrbracket}
\end{aligned}$$

This encoding formally expresses how non-deterministic session protocols (typed with “&”) capture linear and unrestricted resource consumption in $u\widehat{\lambda}_{\oplus}^{\xi}$. Notice that the encoding of the multiset type π depends on two arguments (a strict type σ and a number $i \geq 0$) which are left unspecified above. This is crucial to represent failures in $u\widehat{\lambda}_{\oplus}^{\xi}$ as typable processes

$$\begin{aligned}
 \llbracket \mathbf{unit} \rrbracket &= \& \mathbf{1} \\
 \llbracket \eta \rrbracket &= \&_{\eta_i \in \eta} \{ \mathbf{1}_i; \llbracket \eta_i \rrbracket \} \\
 \llbracket (\sigma^k, \eta) \rightarrow \tau \rrbracket &= \&(\overline{\llbracket (\sigma^k, \eta) \rrbracket}_{(\sigma, i)} \wp \llbracket \tau \rrbracket) \\
 \llbracket (\sigma^k, \eta) \rrbracket_{(\sigma, i)} &= \oplus((\llbracket \sigma^k \rrbracket_{(\sigma, i)} \otimes (!\llbracket \eta \rrbracket) \otimes (\mathbf{1}))) \\
 \llbracket \sigma \wedge \pi \rrbracket_{(\sigma, i)} &= \overline{\&((\oplus \perp) \otimes (\& \oplus ((\& \overline{\llbracket \sigma \rrbracket}) \wp (\overline{\llbracket \pi \rrbracket}_{(\sigma, i)}))))} \\
 &= \oplus((\& \mathbf{1}) \wp (\oplus \&((\oplus \llbracket \sigma \rrbracket) \otimes (\llbracket \pi \rrbracket_{(\sigma, i)})))) \\
 \llbracket \omega \rrbracket_{(\sigma, i)} &= \begin{cases} \overline{\&((\oplus \perp) \otimes (\& \oplus \perp))} & \text{if } i = 0 \\ \overline{\&((\oplus \perp) \otimes (\& \oplus ((\& \overline{\llbracket \sigma \rrbracket}) \wp (\overline{\llbracket \omega \rrbracket}_{(\sigma, i-1)}))))} & \text{if } i > 0 \end{cases}
 \end{aligned}$$

■ **Figure 9** Encoding of intersection types into session types (cf. Def. 34).

in $\mathfrak{s}\pi$. For instance, given $(\sigma^j, \eta) \rightarrow \tau$ and (σ^k, η) , the well-formedness rule for application admits a mismatch ($j \neq k$, see [18]). In our proof of type preservation, the two arguments of the encoding are instantiated appropriately. Notice also how the client-server behaviour of unrestricted resources appears as “! $\llbracket \eta \rrbracket$ ” in the encoding of the tuple type (σ^k, η) . With our encodings of expressions and types in place, we can now define our encoding of judgements:

► **Definition 35.** *If \mathbb{M} is an $u\hat{\lambda}_{\oplus}^{\ddagger}$ expression such that $\Theta; \Gamma \models \mathbb{M} : \tau$ then we define the encoding of the judgement to be: $\llbracket \mathbb{M} \rrbracket_u \vdash \llbracket \Gamma \rrbracket, u : \llbracket \tau \rrbracket; \llbracket \Theta \rrbracket$.*

The correctness of our encoding $\llbracket \cdot \rrbracket_u : u\hat{\lambda}_{\oplus}^{\ddagger} \rightarrow \mathfrak{s}\pi$, stated in Theorem 37 (and detailed in [18]), relies on a notion of *success* for both $u\hat{\lambda}_{\oplus}^{\ddagger}$ and $\mathfrak{s}\pi$, given by the \checkmark construct:

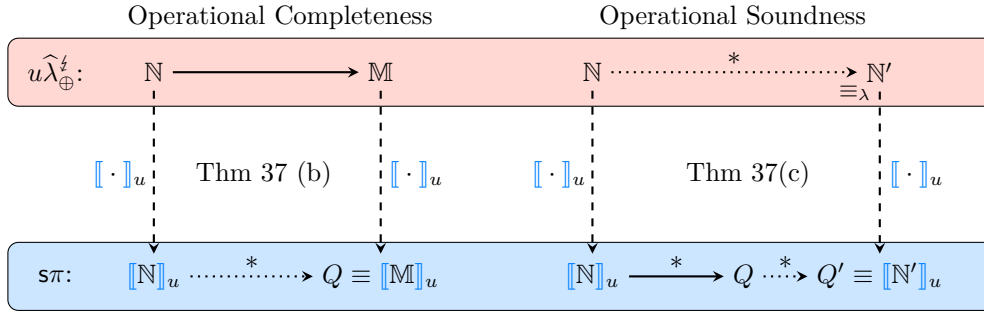
- **Definition 36.** *We extend the syntax of terms for $u\hat{\lambda}_{\oplus}^{\ddagger}$ and processes for $\mathfrak{s}\pi$ with \checkmark :*
- **(In $u\hat{\lambda}_{\oplus}^{\ddagger}$)** $\mathbb{M} \Downarrow_{\checkmark}$ *iff there exist M_1, \dots, M_k such that $\mathbb{M} \rightarrow^* M_1 + \dots + M_k$ and $\text{head}(M_j) = \checkmark$, for some $j \in \{1, \dots, k\}$ and term M'_j such that $M_j \equiv_{\lambda} M'_j$.*
 - **(In $\mathfrak{s}\pi$)** $P \Downarrow_{\checkmark}$ *holds whenever there exists a P' such that $P \rightarrow^* P'$ and P' contains an unguarded occurrence of \checkmark (i.e., an occurrence that does not occur behind a prefix).*

We now state operational correctness. Fig. 10 illustrates the relation between completeness and soundness that the encoding satisfies: solid arrows denote reductions assumed, dashed arrows denote the application of $\llbracket \cdot \rrbracket_u$, and dotted arrows denote the existing reductions that can be implied from the results.

We remark that since $u\hat{\lambda}_{\oplus}^{\ddagger}$ satisfies the diamond property, it suffices to consider completeness based on a single reduction ($\mathbb{N} \rightarrow \mathbb{M}$). Soundness uses the congruence \equiv_{λ} in Def. 27. We write $N \rightarrow_{\equiv_{\lambda}} N'$ iff $N \equiv_{\lambda} N_1 \rightarrow N_2 \equiv_{\lambda} N'$, for some N_1, N_2 . Then, $\rightarrow_{\equiv_{\lambda}}^*$ is the reflexive, transitive closure of $\rightarrow_{\equiv_{\lambda}}$. For success sensitivity, we decree $\llbracket \checkmark \rrbracket_u = \checkmark$. We have:

► **Theorem 37 (Operational Correctness).** *Let \mathbb{N} and \mathbb{M} be well-formed $u\hat{\lambda}_{\oplus}^{\ddagger}$ closed expressions.*

- (a) *(Type Preservation) Let B be a bag. We have:*
- (i) *If $\Theta; \Gamma \models B : (\sigma^k, \eta)$ then $\llbracket B \rrbracket_u \vdash \llbracket \Gamma \rrbracket, u : \llbracket (\sigma^k, \eta) \rrbracket_{(\sigma, i)}; \llbracket \Theta \rrbracket$.*
 - (ii) *If $\Theta; \Gamma \models \mathbb{M} : \tau$ then $\llbracket \mathbb{M} \rrbracket_u \vdash \llbracket \Gamma \rrbracket, u : \llbracket \tau \rrbracket; \llbracket \Theta \rrbracket$.*
- (b) *(Completeness) If $\mathbb{N} \rightarrow \mathbb{M}$ then there exists Q such that $\llbracket \mathbb{N} \rrbracket_u \rightarrow^* Q \equiv_{\lambda} \llbracket \mathbb{M} \rrbracket_u$.*
- (c) *(Soundness) If $\llbracket \mathbb{N} \rrbracket_u \rightarrow^* Q$ then $Q \rightarrow^* Q'$, $\mathbb{N} \rightarrow_{\equiv_{\lambda}}^* \mathbb{N}'$ and $\llbracket \mathbb{N}' \rrbracket_u \equiv Q'$, for some Q', \mathbb{N}' .*
- (d) *(Success Sensitivity) $\mathbb{M} \Downarrow_{\checkmark}$ if, and only if, $\llbracket \mathbb{M} \rrbracket_u \Downarrow_{\checkmark}$.*



■ **Figure 10** An overview of operational soundness and completeness for $\llbracket \cdot \rrbracket_u$.

Proof. All items are proven by structural induction; a detailed proof can be found in [18]. Below we present the most interesting case in the proof of *soundness*: the case when $\mathbb{N} = M(C \star U)$. Then,

$$\llbracket \mathbb{N} \rrbracket_u = \llbracket M(C \star U) \rrbracket_u = \bigoplus_{C_i \in \text{PER}(C)} (\nu v)(\llbracket M \rrbracket_v \mid v.\text{some}_{u, \text{fv}(C)}; \bar{v}(x).([v \leftrightarrow u] \mid \llbracket C_i \star U \rrbracket_x)).$$

The proof then proceeds by induction on the number of reduction steps k that can be taken from $\llbracket \mathbb{N} \rrbracket_u$, i.e., $\llbracket \mathbb{N} \rrbracket_u \rightarrow^k Q$. We will consider the case when $k \geq 1$, where for some process R and non-negative integers n, m such that $k = n + m$, we have the following:

$$\llbracket \mathbb{N} \rrbracket_u \rightarrow^m \bigoplus_{C_i \in \text{PER}(C)} (\nu v)(R \mid v.\text{some}_{u, \text{fv}(C)}; \bar{v}(x).([v \leftrightarrow u] \mid \llbracket C_i \star U \rrbracket_x)) \rightarrow^n Q$$

There are several cases to analyse depending on the values of m and n , and the shape of M . We consider $m = 0, n \geq 1$ and $M = (\lambda x.(M'[\tilde{x} \leftarrow x])) \langle N_1/y_1 \rangle \cdots \langle N_p/y_p \rangle \llbracket U_1/z_1 \rrbracket \cdots \llbracket U_q/z_q \rrbracket$, where $p, q \geq 0$. Then, $\llbracket \mathbb{N} \rrbracket_u$ can perform the following reduction:

$$\llbracket \mathbb{N} \rrbracket_u \rightarrow^* \bigoplus_{C_i \in \text{PER}(C)} (\nu \tilde{y}, \tilde{z}, x)(x.\overline{\text{some}}; x(x^\ell).x(x^1).x[]; \llbracket M'[\tilde{x} \leftarrow x] \rrbracket_u \mid Q'' \mid \llbracket C_i \star U \rrbracket_x) \quad (:= Q_3)$$

where Q'' defines the encoding of explicit substitutions within the encoded subterm M . Notice that:

$$\begin{aligned} \mathbb{N} &= (\lambda x.(M'[\tilde{x} \leftarrow x])) \langle N_1/y_1 \rangle \cdots \langle N_p/y_p \rangle \llbracket U_1/z_1 \rrbracket \cdots \llbracket U_q/z_q \rrbracket (C \star U) \\ &\equiv_\lambda (\lambda x.(M'[\tilde{x} \leftarrow x])) (C \star U) \langle N_1/y_1 \rangle \cdots \langle N_p/y_p \rangle \llbracket U_1/z_1 \rrbracket \cdots \llbracket U_q/z_q \rrbracket \\ &\rightarrow M'[\tilde{x} \leftarrow x] \langle \langle C \star U \rangle / x \rangle \langle N_1/y_1 \rangle \cdots \langle N_p/y_p \rangle \llbracket U_1/z_1 \rrbracket \cdots \llbracket U_q/z_q \rrbracket = \mathbb{M} \end{aligned}$$

where the congruence holds assuming the necessary α -renaming of variables. Finally, one can verify that $\llbracket \mathbb{M} \rrbracket_u = Q_3$, and the result follows. ◀

► **Example 38.** Recall again term M_1 from Example 3. It can be shown that $(M_1)^\bullet \rightarrow^* (\langle N \rangle)^\bullet \llbracket \langle U \rangle / x^1 \rrbracket$. To illustrate operational completeness, we can verify preservation of reduction, via $\llbracket \cdot \rrbracket$: reductions below use the rules for $s\pi$ in Figure 5 – see Figure 11.

5 Concluding Remarks

Summary. We have extended the line of work we developed in [17], on resource λ -calculi with firm logical foundations via typed concurrent processes. We presented $u\lambda_{\oplus}^\xi$, a resource calculus with non-determinism and explicit failures, with dedicated treatment for linear

$$\begin{aligned}
 & \llbracket (M_1)^\bullet \rrbracket = \\
 & (\nu v)(v.\overline{\text{some}}; v(x).x.\overline{\text{some}}; x(x^\ell).x(x').x[]; x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\text{some}_\emptyset; y_1[]; \mathbf{0} | \\
 & \quad x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; x^\ell(x_1).x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2).(y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_v | x^\ell.\overline{\text{none}})) | \\
 & \quad v.\text{some}_{u,\text{fv}(\langle N \rangle^\bullet)}; \overline{v}(x).([v \leftrightarrow u] | x.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \overline{x}(x^\ell).(x^\ell.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; x^\ell(y_1). \\
 & \quad x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}); x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | y_1.\overline{\text{none}} | x^\ell.\text{some}_\emptyset; \\
 & \quad x^\ell(y_2).(y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | \overline{x}(x').(!x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i} | \overline{x}[]))) \\
 & \longrightarrow^3 (\nu x)(x.\overline{\text{some}}; x(x^\ell).x(x').x[]; x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\text{some}_\emptyset; y_1[]; \mathbf{0} | x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; \\
 & \quad x^\ell(x_1).x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2).(y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_u | x^\ell.\overline{\text{none}})) | (x.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \overline{x}(x^\ell). \\
 & \quad (x^\ell.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; x^\ell(y_1).x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}); x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | \\
 & \quad y_1.\overline{\text{none}} | x^\ell.\text{some}_\emptyset; x^\ell(y_2).(y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | \overline{x}(x').(!x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i} | \overline{x}[]))) \\
 & \longrightarrow^2 (\nu x, x^\ell)(x(x').x[]; x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\text{some}_\emptyset; y_1[]; \mathbf{0} | x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; x^\ell(x_1). \\
 & \quad x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2).(y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_u | x^\ell.\overline{\text{none}})) | (x^\ell.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; x^\ell(y_1). \\
 & \quad x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}); x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | y_1.\overline{\text{none}} | x^\ell.\text{some}_\emptyset; x^\ell(y_2). \\
 & \quad (y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | \overline{x}(x').(!x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i} | \overline{x}[]))) \\
 & \longrightarrow (\nu x, x^\ell, x')(x[]; x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\text{some}_\emptyset; y_1[]; \mathbf{0} | x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; x^\ell(x_1).x^\ell.\overline{\text{some}}. \\
 & \quad \overline{x^\ell}(y_2).(y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_u | x^\ell.\overline{\text{none}})) | (x^\ell.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; x^\ell(y_1).x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}); \\
 & \quad x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | y_1.\overline{\text{none}} | \\
 & \quad x^\ell.\text{some}_\emptyset; x^\ell(y_2).(y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | !x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i} | \overline{x}[]))) \\
 & \longrightarrow (\nu x^\ell, x')(x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_1).(y_1.\text{some}_\emptyset; y_1.[]; \mathbf{0} | x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; x^\ell(x_1). \\
 & \quad x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2).(y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_u | x^\ell.\overline{\text{none}})) | (x^\ell.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; x^\ell(y_1). \\
 & \quad x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}); x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | y_1.\overline{\text{none}} | \\
 & \quad x^\ell.\text{some}_\emptyset; x^\ell(y_2).(y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | !x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i}))) \\
 & \longrightarrow (\nu x^\ell, y_1, x')(y_1.\text{some}_\emptyset; y_1.[]; \mathbf{0} | x^\ell.\overline{\text{some}}; x^\ell.\text{some}_u; x^\ell(x_1).x^\ell.\overline{\text{some}}.\overline{x^\ell}(y_2). \\
 & \quad (y_2.\text{some}_{u,x_1}; y_2.[]; \llbracket x_1 \rrbracket_u | x^\ell.\overline{\text{none}})) | (x^\ell.\text{some}_{y_1,\text{fv}(\langle N \rangle^\bullet)}; x^\ell.\overline{\text{some}};\overline{x^\ell}(x_1).(x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \\
 & \quad \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | y_1.\overline{\text{none}} | x^\ell.\text{some}_\emptyset; x^\ell(y_2).(y_2.\overline{\text{some}}; \overline{y_2}[] | x^\ell.\text{some}_\emptyset; x^\ell.\overline{\text{none}})) | !x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i}))) \\
 & \longrightarrow^* (\nu x_1, x')(x_1.\overline{\text{some}}; [x_1 \leftrightarrow u] | x_1.\text{some}_{\text{fv}(\langle N \rangle^\bullet)}; \llbracket \langle N \rangle^\bullet \rrbracket_{x_1} | !x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i}) \\
 & \longrightarrow^* (\nu x')(\llbracket \langle N \rangle^\bullet \rrbracket_u | !x'(x_i).\llbracket \langle U \rangle^\bullet \rrbracket_{x_i}) \\
 & = \llbracket \langle N \rangle^\bullet \llbracket \langle U \rangle^\bullet / x' \rrbracket_u \rrbracket
 \end{aligned}$$

■ **Figure 11** Illustrating operational correspondence, following Example 38.

and unrestricted resources. By means of examples, we illustrated the expressivity, (lazy) semantics, and design decisions underpinning $u\lambda_{\oplus}^{\zeta}$, and introduced a class of well-formed expressions based on intersection types, which includes fail-prone expressions. To bear witness to the logical foundations of $u\lambda_{\oplus}^{\zeta}$, we defined and proved correct a typed encoding into the concurrent calculus $\mathfrak{s}\pi$, which subsumes the one in [17]. We plan to study key properties for $u\lambda_{\oplus}^{\zeta}$ (such as solvability and normalisation) by leveraging our typed encoding into $\mathfrak{s}\pi$.

Related Work. With respect to previous resource calculi, a distinctive feature of $u\lambda_{\oplus}^{\zeta}$ is its support of explicit failures, which may arise depending on the interplay between (i) linear and unrestricted occurrences of variables in a term and (ii) associated resources in the bag. This feature allows $u\lambda_{\oplus}^{\zeta}$ to express variants of usual λ -terms (\mathbf{I} , Δ , Ω) not expressible in other resource calculi.

Related to $u\lambda_{\oplus}^{\zeta}$ is Boudol's work on a λ -calculus in which multiplicities can be infinite [1, 3]. An intersection type system is used to prove *adequacy* with respect to a testing semantics. However, failing behaviours as well as typability are not explored. Multiplicities can be expressed in $u\lambda_{\oplus}^{\zeta}$: a linear resource is available m times when the linear bag contains m copies of it; the term fails if the corresponding number of linear variables is different from m .

Also related is the resource λ -calculus by Pagani and Ronchi della Rocca [16], which includes linear and reusable resources; the latter are available in multisets, also called bags. In their setting, $M[N^!]$ denotes an application of a term M to a resource N that can be used *ad libitum*. Standard terms such as \mathbf{I} , Δ and Ω are expressed as $\lambda x.x$, $\Delta := \lambda x.x[x^!]$, and $\Omega := \Delta[\Delta^!]$, respectively; different variants are possible but cannot express the desired behaviour. A lazy reduction semantics is based on *baby* and *giant* steps: whereas the first consume one resource at each time, the second comprises several baby steps; combinations of the use of resources (by permuting resources in bags) are considered. A (non-idempotent) intersection type system is proposed: normalisation and a characterisation of solvability are investigated. Unlike our work, encodings into the π -calculus are not explored in [16].

References

- 1 Gérard Boudol. The lambda-calculus with multiplicities (abstract). 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 1–6. Springer, 1993. doi:10.1007/3-540-57208-2_1.
- 2 Gérard Boudol and Cosimo Laneve. The discriminating power of multiplicities in the lambda-calculus. *Inf. Comput.*, 126(1):83–102, 1996. doi:10.1006/inco.1996.0037.
- 3 Gérard Boudol and Cosimo Laneve. lambda-calculus, multiplicities, and the pi-calculus. In *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 659–690, 2000.
- 4 Antonio Bucciarelli, Delia Kesner, and Daniel Ventura. Non-idempotent intersection types for the lambda-calculus. *Logic Journal of the IGPL*, 25(4):431–464, 2017.
- 5 Luís Caires and Jorge A. Pérez. Linearity, control effects, and behavioral types. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, volume 10201 of *Lecture Notes in Computer Science*, pages 229–259. Springer, 2017. doi:10.1007/978-3-662-54434-1_9.
- 6 Luís Caires and Frank Pfenning. Session types as intuitionistic linear propositions. In *CONCUR 2010 - Concurrency Theory, 21th International Conference, CONCUR 2010, Paris, France, August 31-September 3, 2010. Proceedings*, pages 222–236, 2010. doi:10.1007/978-3-642-15375-4_16.
- 7 Maurizio Dominici, Simona Ronchi Della Rocca, and Paolo Tranquilli. Standardization in resource lambda-calculus. In *Proceedings 2nd International Workshop on Linearity, LINEARITY 2012, Tallinn, Estonia, 1 April 2012.*, pages 1–11, 2012. doi:10.4204/EPTCS.101.1.
- 8 Silvia Ghilezan, Jelena Ivetic, Pierre Lescanne, and Silvia Likavec. Intersection types for the resource control lambda calculi. In *Theoretical Aspects of Computing - ICTAC 2011 - 8th International Colloquium, Johannesburg, South Africa, August 31 - September 2, 2011. Proceedings*, pages 116–134, 2011. doi:10.1007/978-3-642-23283-1_10.

- 9 Daniele Gorla. Towards a unified approach to encodability and separation results for process calculi. *Inf. Comput.*, 208(9):1031–1053, 2010. doi:10.1016/j.ic.2010.05.002.
- 10 Tom Gundersen, Willem Heijltjes, and Michel Parigot. Atomic lambda calculus: A typed lambda-calculus with explicit sharing. In *28th Annual ACM/IEEE Symposium on Logic in Computer Science, LICS 2013, New Orleans, LA, USA, June 25-28, 2013*, pages 311–320, 2013. doi:10.1109/LICS.2013.37.
- 11 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.
- 12 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.
- 13 Delia Kesner and Stéphane Lengrand. Resource operators for lambda-calculus. *Inf. Comput.*, 205(4):419–473, 2007. doi:10.1016/j.ic.2006.08.008.
- 14 Dimitrios Kouzapas, Jorge A. Pérez, and Nobuko Yoshida. On the relative expressiveness of higher-order session processes. *Inf. Comput.*, 268, 2019. doi:10.1016/j.ic.2019.06.002.
- 15 Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, I. *Inf. Comput.*, 100(1):1–40, 1992. doi:10.1016/0890-5401(92)90008-4.
- 16 Michele Pagani and Simona Ronchi Della Rocca. Solvability in resource lambda-calculus. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures, 13th International Conference, FOSSACS 2010, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2010, Paphos, Cyprus, March 20-28, 2010. Proceedings*, volume 6014 of *Lecture Notes in Computer Science*, pages 358–373. Springer, 2010. doi:10.1007/978-3-642-12032-9_25.
- 17 Joseph W. N. Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez. Non-deterministic functions as non-deterministic processes. 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 21:1–21:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021. doi:10.4230/LIPICs.FSCD.2021.21.
- 18 Joseph W. N. Paulus, Daniele Nantes-Sobrinho, and Jorge A. Pérez. Types and Terms Translated: Unrestricted Resources in Encoding Functions as Processes (Extended Version). *CoRR*, abs/2112.01593, 2021. arXiv:2112.01593.
- 19 Philip Wadler. Propositions as sessions. In Peter Thiemann and Robby Bruce Findler, editors, *ACM SIGPLAN International Conference on Functional Programming, ICFP'12, Copenhagen, Denmark, September 9-15, 2012*, pages 273–286. ACM, 2012. doi:10.1145/2364527.2364568.

Size-Based Termination for Non-Positive Types in Simply Typed Lambda-Calculus

Yuta Takahashi  

Ochanomizu University, Tokyo, Japan

Abstract

So far, several typed lambda-calculus systems were combined with algebraic rewrite rules, and the termination (in other words, strong normalisation) problem of the combined systems was discussed. By the size-based approach, Blanqui formulated a termination criterion for simply typed lambda-calculus with algebraic rewrite rules which guarantees, in some specific cases, the termination of the rewrite relation induced by beta-reduction and algebraic rewrite rules on strictly or non-strictly positive inductive types. Using the inflationary fixed-point construction, we extend this termination criterion so that it is possible to show the termination of the rewrite relation induced by some rewrite rules on types which are called non-positive types. In addition, we note that a condition in Blanqui's proof can be dropped, and this improves the criterion also for non-strictly positive inductive types.

2012 ACM Subject Classification Theory of computation → Equational logic and rewriting; Theory of computation → Type theory

Keywords and phrases termination, higher-order rewriting, non-positive types, inductive types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2021.12

Funding *Yuta Takahashi*: This work is supported by JSPS KAKENHI Grant Number JP21K12822.

Acknowledgements I want to thank Frédéric Blanqui and Ralph Matthes for valuable discussions on size-based termination and non-strictly positive inductive types, respectively. I also thank the anonymous reviewers for their comments which improved the earlier version of this paper.

1 Introduction

1.1 Background

Since the works [12, 21], several typed λ -calculus systems were combined with algebraic rewrite rules, and the termination (i.e., strong normalisation) problem of the combined systems was discussed: for instance, simply typed λ -calculus [11, 15, 18, 9], polymorphic λ -calculus [12, 21, 17], $\lambda\Pi$ -calculus [10], the Calculus of Constructions [25, 7, 8], λ -cube [4], pure type systems [5, 6]. Rewrite rules can make each of these systems more expressive and efficient, and a termination criterion for a combined system provides a sufficient condition for the termination of the rewrite relation (i.e., the reduction relation) in this system. Of course, there are several non-terminating and interesting combined systems, but here we are interested in terminating systems only.

The performance of a combined system depends on not only its type discipline but also the range of rewrite rules whose termination is guaranteed. For instance, while Jouannaud-Okada's work [17] handles polymorphic λ -calculus and Blanqui-Jouannaud-Okada's work [11] does not, the termination criterion in the latter shows the termination of the recursion principle for the Brouwer ordinal type, which cannot be shown by the criterion in the former. The Brouwer ordinal type is a type of well founded trees and a typical example of strictly positive inductive types. Later, Blanqui ([9]) extended the criterion in [11] so that non-strictly positive inductive types can be dealt with. Though the setting of [11, 9] is simply typed λ -calculus, the termination criteria in [11, 9] are powerful enough to deal with several inductive types which are discussed in the literature on type theory.



© Yuta Takahashi;

licensed under Creative Commons License CC-BY 4.0

27th International Conference on Types for Proofs and Programs (TYPES 2021).

Editors: Henning Basold, Jesper Cockx, and Silvia Ghilezan; Article No. 12; pp. 12:1–12:23

Leibniz International Proceedings in Informatics



LIPICs Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

1.2 Aim

We extend the termination criterion in [9] further by making it possible to verify the termination of the rewrite relation induced by some rewrite rules on types which are called *non-positive types*. When we denote arrow types by $T \Rightarrow U$, a non-positive type in simply typed λ -calculus means a sort (i.e., a basic type) \mathbf{B} with a constructor $c : T_1 \Rightarrow \cdots \Rightarrow T_n \Rightarrow \mathbf{B}$ such that \mathbf{B} occurs in some T_i negatively. As shown in [20, 22, 7], there are some non-positive types such that their recursion principles induce non-termination. This indicates the difficulty in finding a terminating example of recursion principles for non-positive types. However, if one considers rewrite rules which are different from recursion principles, one can think of some rewrite rules on non-positive types whose rewrite relation should be shown to be terminating. It is desirable to extend the criterion in [9] in this respect.

1.3 Approach

The approach of [9] to a termination criterion uses computability predicates with size annotations. Roughly speaking, its termination criterion is formulated in the following way: first, an interpretation \mathbb{I} of sorts is defined, and a *computability predicate* is assigned to each type T by extending this interpretation. A computability predicate is a set of terms which satisfies several desirable properties for the purpose of termination proofs. In this first step, the most crucial task is the construction of \mathbb{I} . For any sort \mathbf{B} , $\mathbb{I}(\mathbf{B})$ is defined by using computability predicates annotated by ordinals: $\mathbb{I}(\mathbf{B})$ is equal to $\sup\{\mathcal{S}_\alpha^{\mathbf{B}} \mid \alpha < \mathfrak{h}\}$ for some limit ordinal \mathfrak{h} and some ordinal-indexed family $(\mathcal{S}_\alpha^{\mathbf{B}})_{\alpha < \mathfrak{h}}$ of computability predicates, where $\mathcal{S}_\alpha^{\mathbf{B}} \subseteq \mathcal{S}_\beta^{\mathbf{B}}$ holds for any α, β with $\alpha \leq \beta$. This kind of ordinal-indexed family of computability predicates is called a *stratification*, and the ordinal α in $\mathcal{S}_\alpha^{\mathbf{B}}$ represents the size of terms in $\mathcal{S}_\alpha^{\mathbf{B}}$. The interpretation \mathbb{I} is extended to all types by defining $\mathbb{I}^*(\mathbf{B}) := \mathbb{I}(\mathbf{B})$ and

$$\mathbb{I}^*(T \Rightarrow U) := \mathbb{I}^*(T) \Rightarrow^* \mathbb{I}^*(U) := \{t \mid ts \in \mathbb{I}^*(U) \text{ for any } s \in \mathbb{I}^*(T)\}.$$

Next, a termination criterion is presented, and it is shown that if a given rewrite system satisfies the termination criterion, then any term t of type T belongs to the computability predicate $\mathbb{I}^*(T)$ assigned to T ; this implies that every rewrite sequence from t terminates, hence the correctness of the termination criterion is verified.

As explained in [9], the above notion of size is useful for showing the termination of subtraction and division on the natural number type \mathbf{N} :

$$\begin{array}{lll} \text{sub } x \ 0 \rightarrow x & \text{sub } 0 \ y \rightarrow 0 & \text{sub } (\text{s } x)(\text{s } y) \rightarrow \text{sub } x \ y \\ \text{div } 0 \ (\text{s } y) \rightarrow 0 & \text{div } (\text{s } x) \ (\text{s } y) \rightarrow \text{s } (\text{div } (\text{sub } x \ y) \ (\text{s } y)) & \end{array}$$

The termination of the rewrite relation induced by these rules is not straightforward: it is not obvious to guarantee that the argument $\text{sub } x \ y$ of the function call $\text{div } (\text{sub } x \ y) \ (\text{s } y)$ is “smaller than” $\text{s } x$ of $\text{div } (\text{s } x) \ (\text{s } y)$. But the stratification $(\mathcal{S}_\alpha^{\mathbf{N}})_{\alpha < \mathfrak{h}}$ with size annotation to constructors and function symbols enables to assign sizes to terms so that the size of $\text{sub } x \ y$ is not greater than the size of x , and the size of $\text{s } x$ is greater than the size of x by one.

The main obstacle in extending this approach in [9] to non-positive types is as follows. Recall that, roughly speaking, a positive inductive type is a sort \mathbf{B} which occurs in the types of arguments of its constructors only positively. For any interpretation \mathbb{I} of sorts and any type T , let $[\mathbf{B} : \mathcal{X}, \mathbb{I}]^*T$ be the interpretation of T obtained from the sort-interpretation \mathbb{I}' defined as follows: $\mathbb{I}'(\mathbf{C}) := \mathcal{X}$ if $\mathbf{C} = \mathbf{B}$, otherwise $\mathbb{I}'(\mathbf{C}) := \mathbb{I}(\mathbf{C})$. Then, a crucial fact for the method of [9] is that if \mathbf{B} occurs in T only positively, then $[\mathbf{B} : \mathcal{X}_1, \mathbb{I}]^*T \subseteq [\mathbf{B} : \mathcal{X}_2, \mathbb{I}]^*T$ holds

whenever $\mathcal{X}_1 \subseteq \mathcal{X}_2$ holds. This monotonicity property enables one to define a stratification $\mathcal{S}_0 \subseteq \mathcal{S}_1 \subseteq \dots \subseteq \mathcal{S}_a \subseteq \dots$ in the bottom-up way, but this property does not always hold if \mathbf{B} occurs in T negatively.

We remove this obstacle by using the inflationary fixed-point construction ([24]), which does not assume the monotonicity of operators for fixed points as explained in [1]. This construction provides the following obvious monotonicity to any ordinal-indexed family $(\mathcal{S}_c)_{c < \mathfrak{h}}$ of computability predicates: if $\mathfrak{a} \leq \mathfrak{b}$ holds then $\bigcup_{c \leq \mathfrak{a}} (\mathbf{B} : \mathcal{S}_c, \mathbb{I}^*T) \subseteq \bigcup_{c \leq \mathfrak{b}} (\mathbf{B} : \mathcal{S}_c, \mathbb{I}^*T)$ holds, where \mathbf{B} may occur in T negatively. A trade-off is that, for non-positive types, we need to reformulate a size-based termination argument with pre-fixed points only.

Our construction of computability predicates for non-positive types enables us to extend the *accessibility* condition of the termination criterion in [9]. The extended accessibility condition can be explained as follows: let \mathbf{B} be a non-positive type with a constructor $c : T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow \mathbf{B}$, and x be a variable of type T_i in which \mathbf{B} occurs negatively. In addition, suppose that x also occurs in the right-hand side r of some rewrite rule $fl_1 \dots l_n \rightarrow r$. Then, the extended accessibility says that x must occur in some l_j ($1 \leq j \leq n$) and the path from the position of x in l_j to the position of l_j consists of finitely many full applications of some constructors, where an n -ary constructor c is said to be fully applied if c takes n arguments t_1, \dots, t_n . For instance, a variable g satisfies the extended accessibility if $l_j = cg$ holds with $g : \mathbf{B} \Rightarrow \mathbf{B}$ and $c : (\mathbf{B} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{B}$, because we encounter only the full application of c in the path from the position of g in l_j to the position of l_j . It is crucial that, in this example, our accessibility condition permits the type $\mathbf{B} \Rightarrow \mathbf{B}$ of g to include a negative occurrence of \mathbf{B} , which is not permitted by the accessibility condition in [9]. Together with one more revision of the termination criterion in [9], our accessibility condition provides the difference between this criterion and our termination criterion.

In addition, we note that a condition in Blanqui's proof can be dropped, and this improves the criterion with regard to non-strictly positive inductive types such as the one appearing in Hofmann's extract function for the breadth-first traversal of trees (see, e.g., [19]). Specifically, we drop a condition on a typing rule for the computability closure in [9]. This enables us to guarantee the termination of Hofmann's extract function, while it is, to the best of our knowledge, an open question whether the criterion in [9] guarantees the termination of Hofmann's extract function.

To sum up, our contributions are twofold: first, we extend the termination criterion in [9] by means of the inflationary fixed-point construction so that it is possible to show the termination of the rewrite relation induced by some rewrite rules on non-positive types. Second, we also improve this criterion with regard to non-strictly positive inductive types by verifying that a condition of a typing rule for the computability closure in [9] can be dropped.

1.4 Outline

In Section 2, we provide several preliminary definitions, and recall the facts needed in the later sections. Next, in Section 3, computability predicates with size annotations are defined. Finally, in Section 4, we formulate a termination criterion and prove the computability of typed terms with rewrite rules satisfying this criterion.

2 Preliminaries

For any finite sequence \vec{e} of some elements, we denote the length of \vec{e} by $|\vec{e}|$. The empty sequence is denoted by ϵ . Given a non-empty and countable set \mathbb{S} of sorts, we define the set \mathbb{T} of *types* by induction: (1) $\mathbb{S} \subseteq \mathbb{T}$, and (2) if $T, U \in \mathbb{T}$ holds then $T \Rightarrow U \in \mathbb{T}$ holds. The

arrow symbol \Rightarrow is treated as right associative, and we often abbreviate $T_1 \Rightarrow \cdots \Rightarrow T_n \Rightarrow U$ by $\vec{T} \Rightarrow U$ with $|\vec{T}| = n$. The set \mathbb{L} of *terms* is defined as follows: let \mathbb{V} be a countably infinite set of variables, \mathbb{C} be a countable set of constructors, and \mathbb{F} be a countable set of function symbols such that $\mathbb{V}, \mathbb{C}, \mathbb{F}$ are pairwise disjoint. Then, (1) $\mathbb{V} \cup \mathbb{C} \cup \mathbb{F} \subseteq \mathbb{L}$, (2) if $T \in \mathbb{T}$, $x \in \mathbb{V}$ and $t \in \mathbb{L}$ hold then $\lambda x^T t \in \mathbb{L}$ holds, and (3) if $t, u \in \mathbb{L}$ holds then $tu \in \mathbb{L}$ holds. Below we identify two α -equivalent terms. We also adopt Barendregt's variable condition: no variable occurs both as a free one and as a bound one in a term, and all bound variables in a term are distinct. The set of all free variables in a term t is denoted by $\text{FV}(t)$. We treat the term application tu as left associative, and often abbreviate $tu_1 \cdots u_n$ as $t\vec{u}$ with $|\vec{u}| = n$. When \mathcal{X}_1 and \mathcal{X}_2 are sets of terms, we define $\mathcal{X}_1 \Rightarrow^* \mathcal{X}_2 := \{s \in \mathbb{L} \mid st \in \mathcal{X}_2 \text{ for any } t \in \mathcal{X}_1\}$. The powerset of \mathbb{L} is denoted by $\wp(\mathbb{L})$. As usual, a *position* in an expression such as a term is a string of positive integers (see, e.g., [3]). The subexpression of e at position p is denoted by $e|_p$, and we denote by $e[e']_p$ the expression obtained by replacing the subexpression of e at position p with e' . In addition, we denote by $\text{Pos}(e, e')$ the set of all positions p in e' with $e'|_p = e$.

Mappings from $\mathbb{C} \cup \mathbb{F}$ to \mathbb{T} are denoted by Θ and treated as sets of pairs. We often write $s : T$ whenever $(s, T) \in \Theta$, i.e., $\Theta(s) = T$ holds for a given Θ . For any $s \in \mathbb{C} \cup \mathbb{F}$ with $\Theta(s) = T_1 \Rightarrow \cdots \Rightarrow T_n \Rightarrow \mathbb{B}$ for some sort \mathbb{B} , we put $r^s := n$. A *typing environment* is a mapping from a finite set of variables to a set of types. Typing environments are denoted by Γ, Δ and treated as sets of pairs. In this paper, *typing rules* are the ones of simply typed λ -calculus:

$$\frac{(s, T) \in \Theta \cup \Gamma}{\Gamma \vdash s : T} \quad \frac{\Gamma \vdash s : T \Rightarrow U \quad \Gamma \vdash t : T}{\Gamma \vdash st : U} \quad \frac{\Gamma \cup (x, T) \vdash u : U}{\Gamma \vdash \lambda x^T u : T \Rightarrow U}$$

A *substitution* is a mapping θ from \mathbb{V} to \mathbb{L} such that $\text{dom } \theta := \{x \in \mathbb{V} \mid \theta(x) \neq x\}$ is finite. Define $\text{FV}(\theta) := \bigcup \{\text{FV}(\theta(x)) \mid x \in \text{dom } \theta\}$. Any substitution θ is extended to \mathbb{L} by stipulating $\theta(tu) := \theta(t)\theta(u)$ and $\theta(\lambda x^T u) := \lambda x^T \theta(u)$. We write $\theta(t)$ as $t\theta$, and always assume that no bound variable in t belongs to $\text{dom } \theta \cup \text{FV}(\theta)$, by using α -conversion if necessary.

We say that a pair (l, r) of terms is a *rewrite rule* and write it as $l \rightarrow r$ if there are a function symbol $f \in \mathbb{F}$, a finite sequence \vec{l} of terms, a typing environment Δ and a type T such that

- $l = f \vec{l}$, $\text{FV}(r) \subseteq \text{FV}(l)$ and $\Delta \vdash l : T$ hold,
- (Subject Reduction) for any Γ and any U , if $\Gamma \vdash l : U$ holds then $\Gamma \vdash r : U$ holds.

If \mathcal{R} is a set of rewrite rules, we define the *rewrite relation* $\rightarrow_{\mathcal{R}}$ on \mathbb{L} as follows: $t \rightarrow_{\mathcal{R}} s$ holds if and only if $t = u[l\theta]_p$ and $u[r\theta]_p = s$ holds for some term u , some substitution θ , some position p in t and some $l \rightarrow r \in \mathcal{R}$. We define $\rightarrow := \rightarrow_{\mathcal{R}} \cup \rightarrow_{\beta}$, where $t \rightarrow_{\beta} s$ holds if and only if $t = u_0[(\lambda x^T u_1)u_2]_p$ and $u_0[u_1\{(x, u_2)\}]_p = s$ hold for some terms u_0, u_1, u_2 and some position p in u_0 . For any term t and any set \mathcal{X} of terms, define $\rightarrow(t) := \{u \in \mathbb{L} \mid t \rightarrow u\}$ and $\rightarrow(\mathcal{X}) := \bigcup \{\rightarrow(t) \mid t \in \mathcal{X}\}$. We say that \rightarrow is *finitely branching* if $\rightarrow(t)$ is finite for any $t \in \mathbb{L}$. A term t is *normal* if there is no term t' such that $t \rightarrow t'$ holds. We denote by SN the set of all terms t which have no infinite rewrite sequence $t \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots$.

When \leq is a quasi ordering, we write $e_1 \leq e_2$ & $e_2 \leq e_1$ as $e_1 \cong e_2$. Let R, R_1, \dots, R_n be relations. We write $\vec{x}R_{\text{prod}}\vec{y}$ if and only if $|\vec{x}| = |\vec{y}|$ holds and there is an integer i with $x_i R y_i$ and $x_j = y_j$ for any $j \neq i$. We write $\vec{x}(R_1, \dots, R_n)_{\text{lex}}\vec{y}$ if and only if $|\vec{x}|, |\vec{y}| \geq n$ and there is an integer i such that $x_i R_i y_i$ and $x_j = y_j$ holds for any $j < i$.

Hereafter, we suppose that the following is given:

- a non-empty and countable set \mathbb{S} of sorts,
- a countably infinite set \mathbb{V} of variables, a countable set \mathbb{C} of constructors, a countable set \mathbb{F} of functional symbols and a mapping $\Theta : \mathbb{C} \cup \mathbb{F} \rightarrow \mathbb{T}$,
- a set \mathcal{R} of rewrite rules.

► **Definition 1 (Interpretations of Types).** Let $\mathbb{I} : \mathbb{S} \rightarrow \wp(\mathbb{L})$ be a partial function from \mathbb{S} to $\wp(\mathbb{L})$. We define a partial function $\mathbb{I}^* : \mathbb{T} \rightarrow \wp(\mathbb{L})$ as follows:

- $\mathbb{I}^*(\mathbf{B}) := \mathbb{I}(\mathbf{B})$ if $\mathbb{I}(\mathbf{B})$ is defined, otherwise $\mathbb{I}^*(\mathbf{B})$ is undefined.
- $\mathbb{I}^*(T \Rightarrow U) := \mathbb{I}^*(T) \Rightarrow^* \mathbb{I}^*(U)$ if both $\mathbb{I}^*(T)$ and $\mathbb{I}^*(U)$ are defined, otherwise $\mathbb{I}^*(T \Rightarrow U)$ is undefined.

We write $\mathbb{I}_1^*(T) = \mathbb{I}_2^*(S)$ if and only if both $\mathbb{I}_1^*(T)$ and $\mathbb{I}_2^*(S)$ are defined and equal.

For any partial function $\mathbb{I} : \mathbb{S} \rightarrow \wp(\mathbb{L})$, we denote by $[\mathbf{B} : \mathcal{X}, \mathbb{I}]$ the partial function $\mathbb{I}' : \mathbb{S} \rightarrow \wp(\mathbb{L})$ such that

$$\mathbb{I}'(\mathbf{C}) = \begin{cases} \mathcal{X}, & \text{if } \mathbf{C} = \mathbf{B}, \\ \mathbb{I}(\mathbf{C}), & \text{if } \mathbf{C} \neq \mathbf{B} \text{ and } \mathbb{I}(\mathbf{C}) \text{ is defined,} \\ \text{undefined,} & \text{else.} \end{cases}$$

For the purpose of this paper, the distinction of *positive* positions and *negative* positions in a type is crucial. We denote the set of all positions in a type T by $\text{Pos}(T)$.

► **Definition 2.** For any $T \in \mathbb{T}$, we define the sets $\text{Pos}^+(T)$ and $\text{Pos}^-(T)$ by induction:

- $\text{Pos}^+(\mathbf{B}) := \{\epsilon\}$, $\text{Pos}^-(\mathbf{B}) := \emptyset$.
- $\text{Pos}^s(T \Rightarrow U) := \{1p \mid p \in \text{Pos}^{-s}(T)\} \cup \{2p \mid p \in \text{Pos}^s(U)\}$ for each $s \in \{+, -\}$ with $-+ := -$ and $-- := +$.

We call $\text{Pos}^+(T)$ the set of all positive positions in T , and $\text{Pos}^-(T)$ the set of all negative positions in T . Moreover, for any $\mathbf{B} \in \mathbb{S}$ and any $T \in \mathbb{T}$, we define $\text{Pos}^s(\mathbf{B}, T) := \text{Pos}(\mathbf{B}, T) \cap \text{Pos}^s(T)$ for each $s \in \{+, -\}$.

A *non-positive* type is a sort \mathbf{B} with a constructor $\mathbf{c} : T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow \mathbf{B}$ such that for some i ($1 \leq i \leq n$), $\text{Pos}^-(\mathbf{B}, T_i)$ is non-empty.

The following fact is known:

► **Proposition 3 ([7]).** For any sort \mathbf{B} , if \mathbb{I} is defined for any sort occurring in T except \mathbf{B} and $\text{Pos}(\mathbf{B}, T) \subseteq \text{Pos}^+(\mathbf{B}, T)$ holds, then $[\mathbf{B} : \mathcal{X}, \mathbb{I}]^*(T) : \wp(\mathbb{L}) \rightarrow \wp(\mathbb{L})$ is monotone with respect to \mathcal{X} , that is, if $\mathcal{X}_1 \subseteq \mathcal{X}_2$ holds then $[\mathbf{B} : \mathcal{X}_1, \mathbb{I}]^*(T) \subseteq [\mathbf{B} : \mathcal{X}_2, \mathbb{I}]^*(T)$ holds.

The notion of computability predicate we use is a standard one, as the definition below shows. A term t is *neutral* if t has one of the following forms: (1) $x\vec{s}$, (2) $(\lambda x.t)u\vec{s}$, (3) $\mathbf{f} \vec{t}$, where $|\vec{t}| \geq \max\{|\vec{l}| \mid \mathbf{f} \vec{l} \rightarrow r \in \mathcal{R} \text{ for some } r\}$ holds.

► **Definition 4 (Computability Predicates).** A *computability predicate* is a set \mathcal{S} of terms satisfying

- $\mathcal{S} \subseteq \text{SN}$,
- $\rightarrow(\mathcal{S}) \subseteq \mathcal{S}$,
- if t is neutral and $\rightarrow(t) \subseteq \mathcal{S}$ holds, then $t \in \mathcal{S}$ holds.

Note that for any computability predicates \mathcal{X}_1 and \mathcal{X}_2 , $\mathcal{X}_1 \Rightarrow^* \mathcal{X}_2$ is a computability predicate. In Section 3, we will use the following lemma (for a proof, see [9, Lemma 1]) to define computability predicates with size annotations.

► **Lemma 5.** *If \rightarrow is finitely branching and \mathbb{Q} is a non-empty set of computability predicates with (\mathbb{Q}, \subseteq) well ordered, then $\bigcup \mathbb{Q}$ is a computability predicate.*

The definition of computability predicates with size annotations will proceed along the hierarchy of ordinals; for this purpose, we use the notion of stratification defined below. We denote ordinals by $\mathfrak{a}, \mathfrak{b}, \mathfrak{c}, \mathfrak{d}$, and the first uncountable cardinal by \mathfrak{h} .

► **Definition 6 (Stratifications).** *A stratification is an ordinal-indexed family $(\mathcal{S}_\alpha)_{\alpha < \mathfrak{c}}$ of sets of terms for some $\mathfrak{c} \leq \mathfrak{h}$. For any term $t \in \bigcup_{\alpha < \mathfrak{c}} \mathcal{S}_\alpha$, we define the ordinal $o_{\mathcal{S}}(t)$ as the least ordinal \mathfrak{b} such that $t \in \mathcal{S}_\mathfrak{b}$ holds. We say that a stratification $\mathcal{S} = (\mathcal{S}_\alpha)_{\alpha < \mathfrak{c}}$ is monotone if and only if $\mathcal{S}_\alpha \subseteq \mathcal{S}_\mathfrak{b}$ holds for any $\mathfrak{a}, \mathfrak{b}$ with $\mathfrak{a} \leq \mathfrak{b} < \mathfrak{c}$.*

Let $\mathcal{S} = (\mathcal{S}_\alpha)_{\alpha < \mathfrak{h}}$ be a monotone stratification. Since any set of terms is countable, there is a countable ordinal \mathfrak{a} such that $\mathcal{S}_\mathfrak{a} = \mathcal{S}_\mathfrak{c}$ holds for any $\mathfrak{c} > \mathfrak{a}$. We denote the least ordinal satisfying this property by $\mathfrak{m}(\mathcal{S})$. Moreover, let $\mathcal{S} = (\mathcal{S}_\alpha)_{\alpha < \mathfrak{c}}$ be a monotone stratification, and suppose that \mathbb{I} is defined for any sort occurring in T except \mathbf{B} with $\text{Pos}(\mathbf{B}, T) \subseteq \text{Pos}^+(\mathbf{B}, T)$. Then, by Proposition 3, $([\mathbf{B} : \mathcal{S}_\alpha, \mathbb{I}]^*(T))_{\alpha < \mathfrak{c}}$ is a monotone stratification. We denote this monotone stratification by $[\mathbf{B} : \mathcal{S}, \mathbb{I}]^*(T)$.

One can prove the lemma below as in Lemma 3.(1) of [9].

► **Lemma 7.** *Let $\mathcal{S} = (\mathcal{S}_\alpha)_{\alpha < \mathfrak{h}}$ be a monotone stratification such that every \mathcal{S}_α is a computability predicate, and assume that \rightarrow is finitely branching. If $t \in \mathcal{S}_{\mathfrak{m}(\mathcal{S})}$ and $t \rightarrow t'$ hold, then $t' \in \mathcal{S}_{\mathfrak{m}(\mathcal{S})}$ and $o_{\mathcal{S}}(t) \geq o_{\mathcal{S}}(t')$ hold.*

In the rest of this paper, we always assume that a given rewrite relation \rightarrow is finitely branching.

3 Construction of Computability Predicates with Size Annotations

In this section, we define computability predicates with size annotations. Specifically, we first define the notion of size function (Definition 9), which controls the size-information of computability predicates. Next, given arbitrary size functions, we define stratifications by size functions (Definition 10). It is these stratifications that form a family of computability predicates with size annotations, and this family provides each sort with a computability predicate as its interpretation. Then, this interpretation of sorts is extended to all types in a straightforward way.

We fix an arbitrary well founded ordering $<_{\mathbb{S}}$ on \mathbb{S} , and denote the reflexive closure of $<_{\mathbb{S}}$ by $\leq_{\mathbb{S}}$. Therefore, $\leq_{\mathbb{S}}$ is a partial ordering: there is no pair $(\mathbf{B}_1, \mathbf{B}_2)$ of sorts such that $\mathbf{B}_1 \leq_{\mathbb{S}} \mathbf{B}_2$, $\mathbf{B}_2 \leq_{\mathbb{S}} \mathbf{B}_1$ and $\mathbf{B}_1 \neq \mathbf{B}_2$ hold. Since $\leq_{\mathbb{S}}$ is a partial ordering, we can adopt the following definition of inductive types and non-strictly positive inductive types: a sort \mathbf{B} is an *inductive type* if for any constructor $\mathfrak{c} : T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow \mathbf{B}$ of \mathbf{B} and any i with $1 \leq i \leq n$, $\text{Pos}(\mathbf{B}, T_i) \subseteq \text{Pos}^+(\mathbf{B}, T_i)$ holds. An inductive type \mathbf{B} is *strictly positive* if for any constructor $\mathfrak{c} : T_1 \Rightarrow \dots \Rightarrow T_n \Rightarrow \mathbf{B}$ of \mathbf{B} and any argument T_i of \mathfrak{c} with $T_i = T_{i,n_1} \Rightarrow \dots \Rightarrow T_{i,n_j} \Rightarrow U_i$ ($j \geq 0$), \mathbf{B} does not occur in any of $T_{i,n_1}, \dots, T_{i,n_j}$. A *non-strictly positive inductive type* is an inductive type not being strictly positive.

► **Definition 8 (Arguments of Constructors).** *Let $<_{\mathbb{S}}$ be a well founded ordering on sorts, and T_i be the i -th argument of the constructor $\mathfrak{c} : \vec{T} \Rightarrow \mathbf{B}$. Then,*

1. T_i is recursive iff $\text{Pos}(\mathbf{B}, T_i)$ is not empty,
2. T_i is negative iff $\text{Pos}^-(\mathbf{B}, T_i)$ is not empty and for any \mathbf{C} , either $\text{Pos}(\mathbf{C}, T_i)$ is empty or $\mathbf{C} \leq_{\mathbb{S}} \mathbf{B}$ holds,

3. T_i is accessible iff $\text{Pos}(\mathbf{B}, T_i) \subseteq \text{Pos}^+(\mathbf{B}, T_i)$ holds and for any \mathbf{C} , either $\text{Pos}(\mathbf{C}, T_i)$ is empty or $\mathbf{C} \leq_{\mathbb{S}} \mathbf{B}$ holds.

In order to make our stratifications well-defined (Definition 10 below), hereafter we assume that for any sort \mathbf{B} , \mathbf{B} has no constructor which has some non-negative and non-accessible argument. For instance, when $\mathbf{C} >_{\mathbb{S}} \mathbf{B}$ holds, a constructor $c^{\#} : \mathbf{C} \Rightarrow \mathbf{B}$ of a sort \mathbf{B} has a non-negative and non-accessible argument \mathbf{C} . If we admit $c^{\#}$ then there exists an argument T (i.e., \mathbf{C}) of $c^{\#}$ which includes an occurrence of a sort \mathbf{C} with $\mathbf{C} >_{\mathbb{S}} \mathbf{B}$, and this breaks our definition of stratifications by size functions. Therefore, we may assume without loss of generality that for any constructor $c : \vec{T} \Rightarrow \mathbf{B}$, there are natural numbers n^c, p^c, q^c ($n^c, p^c, q^c \geq 0$) satisfying the following (if needed, we permute the arguments of c):

- for any $i \in \{1, \dots, n^c\}$, the i -th argument T_i of c is negative,
- for any $j \in \{n^c + 1, \dots, p^c\}$, the j -th argument T_j of c is accessible and recursive,
- for any $k \in \{p^c + 1, \dots, q^c\}$, the k -th argument T_k of c is accessible and non-recursive, and there is a sort occurring in T_k only positively,
- for any $l \in \{q^c + 1, \dots, r^c\}$, the l -th argument T_l of c is accessible and non-recursive, and there is no sort occurring in T_l only positively.

When $\Theta(c) = \vec{T} \Rightarrow \mathbf{B}$ holds, $\vec{T} \Rightarrow \mathbf{B}$ always has the following structure:

$$\begin{array}{c} \underbrace{T_1 \Rightarrow \dots \Rightarrow T_{n^c}}_{\text{negative arguments}} \Rightarrow \underbrace{T_{n^c+1} \Rightarrow \dots \Rightarrow T_{p^c}}_{\text{accessible and recursive arguments}} \Rightarrow \\ \underbrace{T_{p^c+1} \Rightarrow \dots \Rightarrow T_{q^c} \Rightarrow T_{q^c+1} \Rightarrow \dots \Rightarrow T_{r^c}}_{\text{accessible and non-recursive arguments}} \Rightarrow \mathbf{B}. \end{array}$$

For instance, if $\mathbf{C} <_{\mathbb{S}} \mathbf{B}$ and $\Theta(c) = (\mathbf{B} \Rightarrow \mathbf{C}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{C} \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{B}$ hold, then we have $n^c = 1$, $p^c = 2$, $q^c = 3$ and $r^c = 4$. On the other hand, if $\mathbf{C} <_{\mathbb{S}} \mathbf{B}$ and $\Theta(c) = (\mathbf{C} \Rightarrow \mathbf{B}) \Rightarrow (\mathbf{C} \Rightarrow \mathbf{C}) \Rightarrow \mathbf{B}$ hold, then $n^c = 0$, $p^c = 1$, $q^c = 1$ and $r^c = 2$ hold. Notice that if $p^c = 0$ holds then $n^c = 0$ holds, and similar implications hold for q^c and r^c .

The size-information of stratifications is controlled by *size functions*, which compute an ordinal as the size of $c \vec{t}$ from ordinals attached to t_1, \dots, t_{q^c} as their sizes.

► **Definition 9** (Size Functions). For any constructor $c : \vec{T} \Rightarrow \mathbf{B}$, a size function (Σ^c, \vec{B}^c) for c consists of a function $\Sigma^c : \mathfrak{h}^{q^c} \rightarrow \mathfrak{h}$ and sorts $\mathbf{B}_1^c, \dots, \mathbf{B}_{q^c}^c$ such that

- for any $i \in \{1, \dots, p^c\}$, $\mathbf{B}_i^c = \mathbf{B}$ holds, and
- for any $i \in \{p^c + 1, \dots, q^c\}$, \mathbf{B}_i^c occurs in T_i with $\text{Pos}(\mathbf{B}_i^c, T_i) \subseteq \text{Pos}^+(\mathbf{B}_i^c, T_i)$ and $\mathbf{B}_i^c <_{\mathbb{S}} \mathbf{B}$.

We often denote a size function (Σ^c, \vec{B}^c) by Σ^c . If $\mathbf{B} \in \mathbb{S}$ holds, then we define

- $\mathbb{C}^{\mathbf{B}} := \{(c, \vec{t}, \vec{T}) \mid c \in \mathbb{C}, c : \vec{T} \Rightarrow \mathbf{B}, |\vec{t}| = |\vec{T}|\}$,
- $\mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t) := \{(c, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}} \mid t \rightarrow^* c \vec{t}\}$, where \rightarrow^* is the reflexive and transitive closure of \rightarrow .

Note that for any $(c, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}$, we do not require $t_i : T_i$.

Below we define *stratifications by size functions*, and these stratifications form the interpretation \mathbb{J} of sorts, which assigns a computability predicate to each sort. Notice that we use the inflationary fixed-point construction (see, e.g., [24, 1, 2]) in the case of negative arguments of constructors.

► **Definition 10** (Stratifications by Size Functions). Assume that a size function is provided with each constructor. For any sort \mathbf{B} , we define the stratification $\mathcal{S}^{\mathbf{B}}$ and the value $\mathbb{J}(\mathbf{B})$ of the function \mathbb{J} from \mathbb{S} to $\wp(\mathbb{L})$ by induction on $>_{\mathbb{S}}$. Suppose that $\mathcal{S}^{\mathbf{C}}$ and $\mathbb{J}(\mathbf{C})$ are defined for any sort $\mathbf{C} <_{\mathbb{S}} \mathbf{B}$. We first define $\mathcal{S}_{\mathfrak{a}}^{\mathbf{B}}$ by induction on $\mathfrak{a} \in \mathfrak{h}$: $\mathcal{S}_0^{\mathbf{B}}$ is defined as the set of all terms $t \in \text{SN}$ such that for any $(c, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t)$,

- $p^c = 0$,
- for any $i \in \{1, \dots, q^c\}$, $t_i \in \mathbb{J}^*(T_i)$, and
- $\Sigma^c(o_{\mathcal{S}^{c,1}}(t_1), \dots, o_{\mathcal{S}^{c,q^c}}(t_{q^c})) = 0$, where $\mathcal{S}^{c,i}$ is a stratification $([\mathbb{B}_i^c : \mathcal{S}_a^{\mathbb{B}_i^c}, \mathbb{J}]^*(T_i))_{a < \mathfrak{h}}$ for any $i \in \{1, \dots, q^c\}$.
We abbreviate $o_{\mathcal{S}^{c,1}}(t_1), \dots, o_{\mathcal{S}^{c,q^c}}(t_{q^c})$ as $o_{\mathcal{S}^c}(\vec{t})$.

When $\mathfrak{a} = \mathfrak{b} + 1$ holds, we define $\mathcal{S}_a^{\mathbb{B}}$ as the set of all terms $t \in \text{SN}$ such that for any $(\mathfrak{c}, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbb{B}}(t)$,

- for any $k \in \{1, \dots, n^c\}$, $t_k \in \bigcup_{\mathfrak{c} \leq \mathfrak{b}} [\mathbb{B} : \mathcal{S}_c^{\mathbb{B}}, \mathbb{J}]^*(T_k)$,
- for any $k \in \{n^c + 1, \dots, p^c\}$, $t_k \in [\mathbb{B} : \mathcal{S}_b^{\mathbb{B}}, \mathbb{J}]^*(T_k)$,
- for any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in \mathbb{J}^*(T_i)$, and
- $\Sigma^c(o_{\mathcal{S}^{c,1}}(t_1), \dots, o_{\mathcal{S}^{c,q^c}}(t_{q^c})) \leq \mathfrak{b} + 1$, where
 - $\mathcal{S}^{c,k}$ is a stratification $(\bigcup_{\mathfrak{c} \leq \mathfrak{c}} [\mathbb{B} : \mathcal{S}_c^{\mathbb{B}}, \mathbb{J}]^*(T_k))_{\mathfrak{c} \leq \mathfrak{b}}$ for any $k \in \{1, \dots, n^c\}$,
 - $\mathcal{S}^{c,k}$ is a stratification $([\mathbb{B} : \mathcal{S}_c^{\mathbb{B}}, \mathbb{J}]^*(T_k))_{\mathfrak{c} \leq \mathfrak{b}}$ for any $k \in \{n^c + 1, \dots, p^c\}$ and
 - $\mathcal{S}^{c,k}$ is a stratification $([\mathbb{B}_k^c : \mathcal{S}_c^{\mathbb{B}_k^c}, \mathbb{J}]^*(T_i))_{\mathfrak{c} < \mathfrak{h}}$ for any $k \in \{p^c + 1, \dots, q^c\}$.

As above, we abbreviate $o_{\mathcal{S}^{c,1}}(t_1), \dots, o_{\mathcal{S}^{c,q^c}}(t_{q^c})$ as $o_{\mathcal{S}^c}(\vec{t})$.

When \mathfrak{a} is a limit ordinal, we define $\mathcal{S}_a^{\mathbb{B}} := \bigcup_{\mathfrak{b} < \mathfrak{a}} \mathcal{S}_b^{\mathbb{B}}$. Finally, we put $\mathcal{S}^{\mathbb{B}} := (\mathcal{S}_a^{\mathbb{B}})_{a < \mathfrak{h}}$ and $\mathbb{J}(\mathbb{B}) := \mathcal{S}_{\mathfrak{m}(\mathcal{S}^{\mathbb{B}})}^{\mathbb{B}}$. To justify the definition of $\mathbb{J}(\mathbb{B})$, we show that $\mathcal{S}^{\mathbb{B}}$ is monotone in Lemma 11.(2) below.

Note that in the definition above we used induction on $>_{\mathfrak{S}}$ and subinduction on $\mathfrak{a} \in \mathfrak{h}$. By the hypothesis of subinduction, we can assume in the case of $\mathfrak{a} = \mathfrak{b} + 1$ that $\mathcal{S}_c^{\mathbb{B}}$ is already defined for any $\mathfrak{c} \leq \mathfrak{b}$. For any sort \mathbb{B} , any set \mathcal{X} of terms and any type T , we abbreviate $[\mathbb{B} : \mathcal{X}, \mathbb{J}]^*(T)$ as $[\mathbb{B} : \mathcal{X}]T$, and $t \in \mathbb{J}^*(T)$ as $t \in T$ for any term t . The lemma below shows that for any sort \mathbb{B} , the stratification $\mathcal{S}^{\mathbb{B}}$ is monotone, and each $\mathcal{S}_a^{\mathbb{B}}$ is a computability predicate.

► **Lemma 11.** *The following statements hold.*

1. Let P, Q be two arbitrary sets of triples $(\mathfrak{c}, \vec{t}, \vec{T})$ of a constructor, a finite sequence of terms and a finite sequence of types. If $P \subseteq Q$ holds then $\{t \in \text{SN} \mid \mathbb{C}_{\rightarrow^*}^{\mathbb{B}}(t) \subseteq P\} \subseteq \{t \in \text{SN} \mid \mathbb{C}_{\rightarrow^*}^{\mathbb{B}}(t) \subseteq Q\}$ holds.
2. For any sort \mathbb{B} , $\mathcal{S}^{\mathbb{B}}$ is monotone.
3. For any sort \mathbb{B} and any $\mathfrak{a} < \mathfrak{h}$, $\mathcal{S}_a^{\mathbb{B}}$ is a computability predicate.

Proof.

(1.) Straightforward.

(2.) We show by induction on \mathfrak{b} that if $\mathfrak{a} \leq \mathfrak{b}$ holds then $\mathcal{S}_a^{\mathbb{B}} \subseteq \mathcal{S}_b^{\mathbb{B}}$ holds. When \mathfrak{b} is 0 or a limit ordinal, the assertion is obvious. Let $\mathfrak{b} = \mathfrak{c} + 1$ be the case. First, we show $\mathcal{S}_c^{\mathbb{B}} \subseteq \mathcal{S}_{\mathfrak{c}+1}^{\mathbb{B}}$. Let $t \in \mathcal{S}_c^{\mathbb{B}}$ be the case. By the definition of $\mathcal{S}^{\mathbb{B}}$, $o_{\mathcal{S}^{\mathbb{B}}}(t)$ cannot be a limit ordinal, hence either $t \in \mathcal{S}_0^{\mathbb{B}}$ or $t \in \mathcal{S}_{\mathfrak{c}_0+1}^{\mathbb{B}}$ holds for some ordinal $\mathfrak{c}_0 < \mathfrak{c}$. Here we consider the latter case only (the former case is similar). In this case, there are sets $P_{\mathfrak{c}_0+1}$ and $P_{\mathfrak{c}+1}$ such that for any $\mathfrak{d} \in \{\mathfrak{c}_0 + 1, \mathfrak{c} + 1\}$, we have $\mathcal{S}_{\mathfrak{d}}^{\mathbb{B}} = \{u \in \text{SN} \mid \mathbb{C}_{\rightarrow^*}^{\mathbb{B}}(u) \subseteq P_{\mathfrak{d}}\}$ and $P_{\mathfrak{d}}$ is the set of all $(\mathfrak{c}, \vec{t}, \vec{T})$ such that

- for any $k \in \{1, \dots, n^c\}$, $t_k \in \bigcup_{\mathfrak{c} < \mathfrak{d}-1} [\mathbb{B} : \mathcal{S}_c^{\mathbb{B}}]T_k$,
- for any $k \in \{n^c + 1, \dots, p^c\}$, $t_k \in [\mathbb{B} : \mathcal{S}_{\mathfrak{d}-1}^{\mathbb{B}}]T_k$,
- for any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in T_i$, and
- $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq \mathfrak{d}$.

By the assertion 1. above, it suffices to verify that $P_{c_0+1} \subseteq P_{c+1}$ holds. Assume that $(c, \vec{t}, \vec{T}) \in P_{c_0+1}$ holds. First, we have $t_k \in \bigcup_{b \leq c} [\mathbf{B} : \mathcal{S}_b^{\mathbf{B}}]T_k$ for any $k \in \{1, \dots, n^c\}$ by $c_0 < c$. Moreover, for any $k \in \{n^c + 1, \dots, p^c\}$, we have $\mathcal{S}_{c_0}^{\mathbf{B}} \subseteq \mathcal{S}_c^{\mathbf{B}}$ by IH, hence we have $t_k \in [\mathbf{B} : \mathcal{S}_c^{\mathbf{B}}]T_k$ by $\text{Pos}(\mathbf{B}, T_k) \subseteq \text{Pos}^+(\mathbf{B}, T_k)$ and Proposition 3. It is obvious that $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq c + 1$ holds, so we have $P_{c_0+1} \subseteq P_{c+1}$. Therefore, $\mathcal{S}_c^{\mathbf{B}} \subseteq \mathcal{S}_{c+1}^{\mathbf{B}}$ holds. Then, by IH, one can see that if $\mathfrak{a} \leq c + 1$ holds then $\mathcal{S}_{\mathfrak{a}}^{\mathbf{B}} \subseteq \mathcal{S}_{c+1}^{\mathbf{B}}$ holds.

- (3.) By induction on \mathfrak{a} . Since the case of $\mathfrak{a} = 0$ is similar to the case of successors, we consider the cases of successors and limits only. As we have seen in the proof of the assertion 2. above, there is a set P_{b+1} such that we have $\mathcal{S}_{b+1}^{\mathbf{B}} = \{t \in \text{SN} \mid \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t) \subseteq P_{b+1}\}$. One can show as in [9, Lemma 6] that for any set P of triples of a constructor, a finite sequence of terms and a finite sequence of types, $\{t \in \text{SN} \mid \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t) \subseteq P\}$ is a computability predicate. Therefore, $\mathcal{S}_{b+1}^{\mathbf{B}}$ is a computability predicate. If \mathfrak{a} is limit, then $\mathcal{S}_{\mathfrak{a}}^{\mathbf{B}} = \bigcup_{b < \mathfrak{a}} \mathcal{S}_b^{\mathbf{B}}$ holds, and each $\mathcal{S}_b^{\mathbf{B}}$ with $b < \mathfrak{a}$ is a computability predicate by IH. Since $((\mathcal{S}_b^{\mathbf{B}})_{b < \mathfrak{a}}, \subset)$ is a well ordering by the monotonicity of $\mathcal{S}^{\mathbf{B}}$, $\mathcal{S}_{\mathfrak{a}}^{\mathbf{B}}$ is a computability predicate by Lemma 5 and the assumption that \rightarrow is finitely branching. ◀

Following [1], one can say that $\mathcal{S}_{\mathfrak{m}(\mathcal{S}^{\mathbf{B}})}^{\mathbf{B}}$ is a *pre-fixed point* in the following sense: for any (c, \vec{t}, \vec{T}) such that

- $c : \vec{T} \Rightarrow \mathbf{B}$ holds, $|\vec{t}| = |\vec{T}|$ holds and $c \vec{t}$ is normal,
- for any $k \in \{1, \dots, n^c\}$, $t_k \in \bigcup_{c \leq \mathfrak{m}(\mathcal{S}^{\mathbf{B}})} [\mathbf{B} : \mathcal{S}_c^{\mathbf{B}}]T_k$,
- for any $k \in \{n^c + 1, \dots, p^c\}$, $t_k \in [\mathbf{B} : \mathcal{S}_{\mathfrak{m}(\mathcal{S}^{\mathbf{B}})}^{\mathbf{B}}]T_k$,
- for any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in T_i$, and
- $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq \mathfrak{m}(\mathcal{S}^{\mathbf{B}}) + 1$,

we have $c \vec{t} \in \mathcal{S}_{\mathfrak{m}(\mathcal{S}^{\mathbf{B}})}^{\mathbf{B}}$. Hereafter, we often abbreviate $t \in \mathcal{S}_{\mathfrak{m}(\mathcal{S}^{\mathbf{B}})}^{\mathbf{B}}$ as $t \in \mathcal{S}^{\mathbf{B}}$.

The statements 1–3 of the lemma below are used in the proof of its statement 4, while the statement 4 will be used in the proof of Lemma 13 below.

▶ **Lemma 12.** *The following statements hold:*

1. $t \in \mathcal{S}_0^{\mathbf{B}}$ holds iff $t \in \mathcal{S}^{\mathbf{B}}$ holds and for any $(c, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t)$, $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = p^c = 0$ holds.
2. $t \in \mathcal{S}_{\mathfrak{a}+1}^{\mathbf{B}}$ holds iff
 - $t \in \mathcal{S}^{\mathbf{B}}$ holds,
 - for any $(c, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t)$, $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq \mathfrak{a} + 1$ holds, and for any $k \in \{1, \dots, p^c\}$, $o_{\mathcal{S}^{c,k}}(t_k) \leq \mathfrak{a}$ holds.
3. If $(c, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}$ and $c \vec{t} \in \mathcal{S}^{\mathbf{B}}$ holds, then we have $o_{\mathcal{S}^{\mathbf{B}}}(c \vec{t}) \geq \Sigma^c(o_{\mathcal{S}^c}(\vec{t}))$ and $o_{\mathcal{S}^{\mathbf{B}}}(c \vec{t}) > o_{\mathcal{S}^{c,k}}(t_k)$ for any $k \in \{1, \dots, p^c\}$.
4. Let δ be the function on \mathfrak{h} such that $\delta(\mathfrak{a}) = \mathfrak{a} + 1$ if \mathfrak{a} is a limit ordinal, and $\delta(\mathfrak{a}) = \mathfrak{a}$ otherwise. If $t \in \mathcal{S}^{\mathbf{B}}$ holds, then we have $o_{\mathcal{S}^{\mathbf{B}}}(t) = \delta(\sup(R \cup S \cup T))$ with
 - $R = \{o_{\mathcal{S}^{\mathbf{B}}}(t') \mid t \rightarrow t'\}$,
 - $S = \{o_{\mathcal{S}^{c,k}}(t_k) + 1 \mid (c, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}, t = c \vec{t}, 1 \leq k \leq p^c\}$,
 - $T = \{\Sigma^c(o_{\mathcal{S}^c}(\vec{t}))\}$ with $(c, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}$ and $t = c \vec{t}$.

Proof.

- (1.) (\implies) Obvious. (\impliedby) Since $t \in \mathcal{S}^{\mathbf{B}}$ holds, we have $t \in \mathcal{S}_{\mathfrak{a}}^{\mathbf{B}}$ with $\mathfrak{a} = o_{\mathcal{S}^{\mathbf{B}}}(t)$. Then, by definition, we have $t \in \text{SN}$, and for any $(c, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t)$ and any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in T_i$ holds. Therefore, we have $t \in \mathcal{S}_0^{\mathbf{B}}$ because $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = 0$ holds.
- (2.) (\implies) Obvious. (\impliedby) If $p^c = 0$ holds then we immediately have $t \in \mathcal{S}_{\mathfrak{a}+1}^{\mathbf{B}}$ by assumption. Suppose that $p^c \geq 1$ holds. We have $t \in \mathcal{S}_{c+1}^{\mathbf{B}}$ with $c + 1 = o_{\mathcal{S}^{\mathbf{B}}}(t)$, so $t \in \text{SN}$ holds. If $(c, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow^*}^{\mathbf{B}}(t)$ holds, then for any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in T_i$ holds. Moreover, we have $o_{\mathcal{S}^{c,k}}(t_k) \leq \mathfrak{a}$ for any $k \in \{1, \dots, p^c\}$. Therefore, $t_k \in \bigcup_{b \leq \mathfrak{a}} [\mathbf{B} : \mathcal{S}_b^{\mathbf{B}}]T_k$ holds for any

$k \in \{1, \dots, n^c\}$, and $t_k \in [\mathbf{B} : \mathcal{S}_a^{\mathbf{B}}]T_k$ holds for any $k \in \{n^c + 1, \dots, p^c\}$. Then, $t \in \mathcal{S}_{a+1}^{\mathbf{B}}$ holds because we have $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq a + 1$.

(3.) One can see that $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t})$ is either 0 or a successor $a + 1$. If $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = 0$ holds, then $\mathbf{c} \vec{t} \in \mathcal{S}_0^{\mathbf{B}}$ holds and so we have $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = 0 \geq \Sigma^c(o_{\mathcal{S}^c}(\vec{t}))$ by definition. If $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = a + 1$ holds, then $\mathbf{c} \vec{t} \in \mathcal{S}_{a+1}^{\mathbf{B}}$ holds and so we have $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = a + 1 \geq \Sigma^c(o_{\mathcal{S}^c}(\vec{t}))$ by definition. Next, we show that $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) > o_{\mathcal{S}^{c,k}}(t_k)$ holds for any $k \in \{1, \dots, p^c\}$. If $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = 0$ holds, then $p^c = 0$ holds and so the assertion holds vacuously. Let $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = a + 1$ be the case, and take a natural number $k \in \{1, \dots, p^c\}$. If $k \leq n^c$ holds, then we have $t_k \in \bigcup_{b \leq a} [\mathbf{B} : \mathcal{S}_b^{\mathbf{B}}]T_k$ and so $o_{\mathcal{S}^{c,k}}(t_k) \leq a$ holds. Otherwise we have $t_k \in [\mathbf{B} : \mathcal{S}_a^{\mathbf{B}}]T_k$, hence $o_{\mathcal{S}^{c,k}}(t_k) \leq a$ holds as well.

(4.) We put $\mathbf{a} := \sup(R \cup S \cup T)$ and $\mathbf{b} := o_{\mathcal{S}^{\mathbf{B}}}(t)$. First, we show $\mathbf{b} \geq \delta(\mathbf{a})$. Let $t \rightarrow t'$ be the case, then we have $\mathbf{b} \geq o_{\mathcal{S}^{\mathbf{B}}}(t')$ by Lemma 7. We have $\mathbf{b} \geq \sup(S \cup T)$ by the statement 3. above, hence $\mathbf{b} \geq \mathbf{a}$. Since \mathbf{b} cannot be a limit ordinal, if \mathbf{a} is a limit ordinal then $\mathbf{b} > \mathbf{a}$ holds, so $\mathbf{b} \geq \delta(\mathbf{a})$ holds. Otherwise, we have $\mathbf{b} \geq \mathbf{a} = \delta(\mathbf{a})$.

Next, we show $\delta(\mathbf{a}) \geq \mathbf{b}$. It suffices to show $t \in \mathcal{S}_{\delta(\mathbf{a})}^{\mathbf{B}}$. Since $t \in \mathcal{S}^{\mathbf{B}}$ holds, we have $t \in \text{SN}$, and for any $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow_*}^{\mathbf{B}}(t)$ and any $i \in \{p^c + 1, \dots, q^c\}$, $t_i \in T_i$ holds.

- $\delta(\mathbf{a}) = 0$: let $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow_*}^{\mathbf{B}}(t)$ be the case. By the statement 1. above, it suffices to show $p^c = 0$ and $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = 0$. First, consider the case of $t = \mathbf{c} \vec{t}$. Then, S must be empty and so $p^c = 0$ holds. Moreover, we have $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = 0$ by $\sup(T) = 0$. Next, let $t \rightarrow t' \rightarrow_* \mathbf{c} \vec{t}$ be the case. Then, we have $o_{\mathcal{S}^{\mathbf{B}}}(t') = 0$ and so $t' \in \mathcal{S}_0^{\mathbf{B}}$ holds. It follows that $p^c = 0$ and $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = 0$ hold, because $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow_*}^{\mathbf{B}}(t')$ holds.
- $\delta(\mathbf{a}) = \mathbf{c} + 1$: let $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}_{\rightarrow_*}^{\mathbf{B}}(t)$ be the case. By the statement 2. above, it suffices to show that $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq \mathbf{c} + 1$ holds and $o_{\mathcal{S}^{c,k}}(t_k) \leq \mathbf{c}$ holds for any $k \in \{1, \dots, p^c\}$. Consider the case of $t = \mathbf{c} \vec{t}$ first. We have $\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) = \sup(T) \leq \mathbf{c} + 1$. Moreover, for any $k \in \{1, \dots, p^c\}$, we have $o_{\mathcal{S}^{c,k}}(t_k) + 1 \leq \sup(S) \leq \mathbf{a} \leq \mathbf{c} + 1$. Next, let $t \rightarrow t' \rightarrow_* \mathbf{c} \vec{t}$ be the case. We have $t', \mathbf{c} \vec{t} \in \mathcal{S}^{\mathbf{B}}$ because $\mathcal{S}_{\mathbf{m}(\mathcal{S}^{\mathbf{B}})}^{\mathbf{B}}$ is a computability predicate. By Lemma 7 and the statement 3. above, we have

$$\Sigma^c(o_{\mathcal{S}^c}(\vec{t})) \leq o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) \leq o_{\mathcal{S}^{\mathbf{B}}}(t') \leq \mathbf{c} + 1.$$

For any $k \in \{1, \dots, p^c\}$, we have

$$o_{\mathcal{S}^{c,k}}(t_k) < o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) \leq o_{\mathcal{S}^{\mathbf{B}}}(t') \leq \mathbf{c} + 1$$

by Lemma 7 and the statement 3. again. Therefore, we have $t \in \mathcal{S}_{\delta(\mathbf{a})}^{\mathbf{B}}$. ◀

By using Lemma 12.(4) as in the proof of [9, Corollaries 1 and 2], one can prove the following lemma, which is a key lemma for our termination criterion (Theorem 24 below).

► **Lemma 13.** *Let \mathbf{c} be a constructor, and assume that Σ^c is strictly extensive with respect to recursive arguments of \mathbf{c} , i.e., $\mathbf{a}_i < \Sigma^c(\vec{\mathbf{a}})$ holds for any $\vec{\mathbf{a}}$ and any $i \in \{1, \dots, p^c\}$. Moreover, we suppose that $\Sigma^c(\vec{\mathbf{a}})$ is not a limit ordinal for any $\vec{\mathbf{a}}$.*

1. *For any $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}$ such that $\mathbf{c} \vec{t} \in \mathcal{S}^{\mathbf{B}}$ holds and $\mathbf{c} \vec{t}$ is normal, we have $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = \Sigma^c(o_{\mathcal{S}^c}(\vec{t}))$.*
2. *If Σ^c is monotone, then for any $(\mathbf{c}, \vec{t}, \vec{T}) \in \mathbb{C}^{\mathbf{B}}$ with $\mathbf{c} \vec{t} \in \mathcal{S}^{\mathbf{B}}$, we have $o_{\mathcal{S}^{\mathbf{B}}}(\mathbf{c} \vec{t}) = \Sigma^c(o_{\mathcal{S}^c}(\vec{t}))$.*

4 Computability of Well-Typed Terms

In this section, we first define the notion of annotation for constructors (Definition 17). An annotation for constructors determines the corresponding size functions (Definition 18), so one can define stratifications by size functions and the interpretation \mathbb{J} of sorts along

Definition 10 above whenever an annotation for constructors is given. Next, we define the notion of annotations for function symbols (Definition 20). Finally, we give our termination criterion, namely, a sufficient condition for the termination of \rightarrow which consists of several criteria concerning annotations for constructors and function symbols (Theorem 24).

First of all, we recall *accessible subterms* in [9], and define a variant of these terms called *quasi-accessible subterms*.

► **Definition 14** (Accessible Subterms and Quasi-Accessible Subterms). *Assume that a family $(\Sigma^c, \vec{B}^c)_{c \in \mathbb{C}}$ of size functions is given. We say that a triple (u, U, C) is accessible in a tuple (t, T, B) and write $(u, U, C) \trianglelefteq_a (t, T, B)$ if and only if either 1. or 2. below is satisfied:*

1. $(u, U, C) = (t, T, B)$.
2. *There are a tuple $(c, \vec{t}, \vec{T}) \in \mathbb{C}^B$ and an integer $k \in \{n^c + 1, \dots, q^c\}$ such that $t = c \vec{t}$, $T = B$ and $(u, U, C) \trianglelefteq_a (t_k, T_k, B_k^c)$ hold.*

We say that (u, U, C) is quasi-accessible in (t, T, B) and write $(u, U, C) \trianglelefteq_{qa} (t, T, B)$ if and only if either 1. above or 2' below is satisfied:

- 2'. *There are a tuple $(c, \vec{t}, \vec{T}) \in \mathbb{C}^B$ and an integer $k \in \{1, \dots, q^c\}$ such that $t = c \vec{t}$, $T = B$ and $(u, U, C) \trianglelefteq_{qa} (t_k, T_k, B_k^c)$ hold.*

Note that if $(u, U, C) \trianglelefteq_{qa} (t, T, B)$ holds by the condition 2' above, then either $(u, U, C) \trianglelefteq_a (t, T, B)$ holds or there are finitely many distinct tuples $(u_1, U_1, C_1), \dots, (u_n, U_n, C_n)$ ($n \geq 1$) such that $(u_n, U_n, C_n) \trianglelefteq_a \dots \trianglelefteq_a (u_1, U_1, C_1) = (t, T, B)$ holds with $u_n = c \vec{t}$ and we have $u = t_k$ for some negative argument k of c . In other words, $\trianglelefteq_{qa} \setminus \trianglelefteq_a$ (i.e., the difference of \trianglelefteq_{qa} and \trianglelefteq_a) holds at most once at the beginning. For instance, consider two constructors $c_0 : (B \Rightarrow B) \Rightarrow B$, $c_1 : B \Rightarrow B$ and a variable $x : B \Rightarrow B$. Then, $(x, B \Rightarrow B, B) \trianglelefteq_{qa} (c_1 (c_0 x), B, B)$ holds by the condition 2', and we have $(x, B \Rightarrow B, B) \trianglelefteq_{qa} (c_0 x, B, B) \trianglelefteq_a (c_1 (c_0 x), B, B)$ in this case. As this example shows, if one has $(u, U, C) \trianglelefteq_{qa} (t, T, B)$ or $(u, U, C) \trianglelefteq_a (t, T, B)$ with $(u, U, C) \neq (t, T, B)$, then $T = B$ must hold by definition. That is why $\trianglelefteq_{qa} \setminus \trianglelefteq_a$ holds at most once: if U is a type of some negative argument of a constructor for C and we have $(u, U, C) \trianglelefteq_{qa} (t, T, B)$, then $U \neq C$ must hold.

► **Lemma 15.** *Assume that a family $(\Sigma^c, \vec{B}^c)_{c \in \mathbb{C}}$ of size functions is given. If $(u, U, C) \trianglelefteq_a (t, T, B)$ and $t \in T$ hold then $u \in U$ holds.*

Proof. The assertion is trivial if $(u, U, C) = (t, T, B)$ holds, so assume that there are a tuple $(c, \vec{t}, \vec{T}) \in \mathbb{C}^B$ and an integer $k \in \{n^c + 1, \dots, q^c\}$ with $t = c \vec{t}$, $T = B$ and $(u, U, C) \trianglelefteq_a (t_k, T_k, B_k^c)$. Since $c \vec{t} \in T = B$ holds, we have $c \vec{t} \in \mathcal{S}_a^B$ for some a which is zero or a successor ordinal. If $k > p^c$ holds, then we immediately have $t_k \in T_k$ by definition. If $k \leq p^c$ holds, then $t_k \in [B : \mathcal{S}_b^B]T_k$ holds for some b with $a = b + 1$. By Proposition 3, we have

$$[B : \mathcal{S}_b^B]T_k \subseteq [B : \mathcal{S}_{m(\mathcal{S}^B)}^B]T_k = T_k,$$

so we have $t_k \in T_k$ also in this case. Therefore, we have $u \in U$ by IH. ◀

Size algebras enable us to annotate constructors and function symbols in a way which estimates the sizes of their inputs and outputs.

► **Definition 16** (Size Algebras). *A size algebra consists of*

- *a set $A = T(F, V)$ of F-terms built from a set V of size variables α, β, \dots and a set F of size function symbols $\mathbf{f}, \mathbf{g}, \dots$ of fixed arity with $V \cap F = \emptyset$,*
- *a quasi ordering \leq_A on A and a strict ordering $<_A \subseteq \leq_A$ such that for each $R \in \{\leq_A, <_A\}$ and any substitution $\varphi : V \rightarrow A$, if aRb holds then $a\varphi R b\varphi$ holds,*

12:12 Size-Based Termination for Non-Positive Types in Simply Typed Lambda-Calculus

- a function $\mathfrak{f}_\mathfrak{h} : \mathfrak{h}^n \rightarrow \mathfrak{h}$ for each size function symbol $\mathfrak{f} \in \mathbf{F}$ of arity n such that for any valuation $\mu : \mathbf{V} \rightarrow \mathfrak{h}$, if $a \leq_{\mathbf{A}} b$ (resp. $a <_{\mathbf{A}} b$) holds then $a\mu \leq b\mu$ (resp. $a\mu < b\mu$) holds, where $\alpha\mu := \mu(\alpha)$ and $(\mathfrak{f} a_1 \dots a_n)\mu := \mathfrak{f}_\mathfrak{h}(a_1\mu, \dots, a_n\mu)$.

Size algebras are denoted by \mathbf{A} . A size algebra \mathbf{A} is monotone if and only if for any $\mathfrak{f} \in \mathbf{F}$, if $\vec{a}(\leq_{\mathbf{A}})_{\text{prod}} \vec{b}$ holds then $\mathfrak{f} \vec{a} \leq_{\mathbf{A}} \mathfrak{f} \vec{b}$ holds.

The *successor size algebra* is the size algebra \mathbf{A} which consists of $\mathbf{F}, <_{\mathbf{A}}, \leq_{\mathbf{A}}$ below:

- $\mathbf{F} = \mathbf{C} \cup \{\mathfrak{s}\}$, where \mathbf{C} is a fixed countably infinite set of constants and \mathfrak{s} is the unary function symbol. The interpretation $\mathfrak{s}_\mathfrak{h}$ of \mathfrak{s} is the successor function on ordinals, and each constant in \mathbf{C} is interpreted in a fixed way.
- $<_{\mathbf{A}}$ is defined by induction: $a <_{\mathbf{A}} \mathfrak{s} a$ for any $a \in \mathbf{A}$, and if $a <_{\mathbf{A}} b$ then $\mathfrak{s} a <_{\mathbf{A}} \mathfrak{s} b$. The ordering $\leq_{\mathbf{A}}$ is defined as the reflexive closure of $<_{\mathbf{A}}$.

One can easily see that the successor size algebra actually satisfies the conditions for being a size algebra. We will use the successor size algebra in applying our termination criterion below (Examples 27 and 28).

The set $\mathbb{T}_{\mathbf{A}}$ of *annotated types* is defined by induction: (1) $\mathbb{T} \subseteq \mathbb{T}_{\mathbf{A}}$, (2) if $\mathbf{B} \in \mathbb{S}$ and $a \in \mathbf{A}$ hold then $\mathbf{B}_a \in \mathbb{T}_{\mathbf{A}}$ holds, (3) if $U, T \in \mathbb{T}_{\mathbf{A}}$ holds then $U \Rightarrow T \in \mathbb{T}_{\mathbf{A}}$ holds. We adopt the following notations:

- $\text{Var}(e)$ means the set of all size variables occurring in an expression e ,
- $|T|$ means the type obtained by removing all annotations in T ,
- $\text{Annot}(T, \mathbf{B}, a)$ means the annotated type obtained by annotating any occurrence of \mathbf{B} in T by a .

In addition, we extend the positive and negative positions in types to annotated types:

- $\text{Pos}^s(\mathbf{B}_b) := \{1p \mid p \in \text{Pos}^s(b)\}$ for each $s \in \{+, -\}$,
- $\text{Pos}^+(\alpha) := \{\epsilon\}$, $\text{Pos}^-(\alpha) := \emptyset$,
- if \mathfrak{f} is of arity 0 then we define $\text{Pos}^+(\mathfrak{f}) := \{\epsilon\}$ and $\text{Pos}^-(\mathfrak{f}) := \emptyset$, otherwise we define

$$\text{Pos}^s(\mathfrak{f} b_1 \dots b_n) := \{ip \mid i \in \text{Mon}^+(\mathfrak{f}), p \in \text{Pos}^s(b_i)\} \cup \{ip \mid i \in \text{Mon}^-(\mathfrak{f}), p \in \text{Pos}^{-s}(b_i)\},$$

where $\text{Mon}^+(\mathfrak{f})$ (resp. $\text{Mon}^-(\mathfrak{f})$) is the set of arguments in which \mathfrak{f} is monotone (resp. anti-monotone) with respect to $\leq_{\mathbf{A}}$.

The *top-extension* of a size algebra \mathbf{A} is the set $\bar{\mathbf{A}} = \mathbf{A} \cup \{\infty\}$ with $\infty \notin \mathbf{A}$. We define as follows:

- $\mathbf{B}_\infty := \mathbf{B}$ for any $\mathbf{B} \in \mathbb{S}$.
- For any $a, b \in \bar{\mathbf{A}}$, $a \leq_{\bar{\mathbf{A}}}^\infty b$ holds if and only if either $a \leq_{\mathbf{A}} b$ or $b = \infty$ holds. In addition, $a <_{\bar{\mathbf{A}}}^\infty b$ holds if and only if either $a <_{\mathbf{A}} b$ or $a \neq \infty = b$ holds.
- For any substitution $\varphi : \mathbf{V} \rightarrow \bar{\mathbf{A}}$ and any $a \in \bar{\mathbf{A}}$, $a\varphi := \infty$ if there is a variable $\alpha \in \text{Var}(a)$ with $\varphi(\alpha) = \infty$, otherwise $a\varphi$ is defined as the usual substitution.

Below we denote elements of $\mathbf{V} \cup \{\infty\}$ by α, β, \dots as well. Then, *annotations for constructors* can be formulated as follows:

► **Definition 17** (Annotations for Constructors). *Let \mathbf{A} be an arbitrary size algebra. An annotation for constructors is a family $C = (\vec{\mathbf{B}}^c, \vec{\Theta}(c))_{c \in \mathbb{C}}$ such that for any $c \in \mathbb{C}$ with $\Theta(c) = T_1 \Rightarrow \dots \Rightarrow T_{r^c} \Rightarrow \mathbf{B}$,*

1. $\vec{\mathbf{B}}^c$ consists of the sorts $\mathbf{B}_1^c, \dots, \mathbf{B}_{q^c}^c$, and
 - for any $i \in \{1, \dots, p^c\}$, $\mathbf{B}_i^c = \mathbf{B}$ holds, and
 - for any $i \in \{p^c + 1, \dots, q^c\}$, \mathbf{B}_i^c occurs in T_i with $\text{Pos}(\mathbf{B}_i^c, T_i) \subseteq \text{Pos}^+(\mathbf{B}_i^c, T_i)$ and $\mathbf{B}_i^c <_{\mathbb{S}} \mathbf{B}$,
2. $\vec{\Theta}(c) = \vec{T}_1 \Rightarrow \dots \Rightarrow \vec{T}_{r^c} \Rightarrow \mathbf{B}_{\sigma^c}$ with $\sigma^c \in \bar{\mathbf{A}}$, where $\vec{T}_i = \text{Annot}(T_i, \mathbf{B}_i^c, \alpha_i^c)$ if $i \in \{1, \dots, q^c\}$, otherwise $\vec{T}_i = T$.

3. $\{\alpha_1^c, \dots, \alpha_{p^c}^c\} \subseteq \mathbb{V}$, $\{\alpha_{p^c+1}^c, \dots, \alpha_{q^c}^c\} \subseteq \mathbb{V} \cup \{\infty\}$, and the members of $\{\alpha_1^c, \dots, \alpha_{q^c}^c\} \cap \mathbb{V}$ are either pairwise equal or pairwise distinct,
4. if $p^c \neq 0$ then $\text{Var}(\sigma^c) \subseteq \{\alpha_1, \dots, \alpha_{q^c}\}$ holds, otherwise $\sigma^c \in \mathbb{V} \setminus \{\alpha_1, \dots, \alpha_{q^c}\}$ holds,
5. $\text{Pos}(\alpha_i^c, \sigma^c) \subseteq \text{Pos}^+(\alpha_i^c, \sigma^c)$ for any $i \in \{1, \dots, q^c\}$,
6. $\alpha_i^c <_{\mathbb{A}}^{\infty} \sigma^c$ holds for any $i \in \{1, \dots, p^c\}$.

We stipulate that an annotation for constructors determines the corresponding size functions in the following way:

► **Definition 18** (Size Functions by Annotation). *Let an annotation for constructors be given. For any constructor c , we define the size function Σ^c induced by this annotation as follows: put $\vec{\alpha} := \alpha_1^c, \dots, \alpha_{q^c}^c$, and suppose that $\mathbf{a}_1, \dots, \mathbf{a}_{q^c}$ are arbitrary ordinals less than \mathfrak{h} . For any $\alpha \in \text{Var}(\sigma^c)$, we first define the valuation ν as*

$$\nu(\alpha) := \begin{cases} 0, & \text{if } \alpha \text{ is distinct from any of } \vec{\alpha}, \\ \mathbf{a}_i, & \text{if } \alpha = \alpha_i^c \text{ and the members of } \{\vec{\alpha}\} \cap \mathbb{V} \text{ are pairwise distinct,} \\ \mathbf{b}, & \text{if } \alpha = \alpha_i^c \text{ and the members of } \{\vec{\alpha}\} \cap \mathbb{V} \text{ are pairwise equal,} \end{cases}$$

where $\mathbf{b} = \sup\{\mathbf{a}_i \mid 1 \leq i \leq q^c, \alpha_i^c \in \mathbb{V}\}$. Then, we define

$$\Sigma^c(\mathbf{a}_1, \dots, \mathbf{a}_{q^c}) := \begin{cases} 0, & \text{if } \sigma^c = \infty, \\ \sigma^c \nu, & \text{otherwise.} \end{cases}$$

When an annotation of constructors is given, we have the size function Σ^c for any constructor c by the definition above. Then, by Definition 10, we obtain stratifications by size functions. By using these stratifications with a given valuation $\mu : \mathbb{V} \rightarrow \mathfrak{h}$, we interpret annotated types $T \in \mathbb{T}_{\mathbb{A}}$ as follows:

- $B\mu := \mathcal{S}_{\mathfrak{m}(S^{\mathbb{B}})}^{\mathbb{B}}$, and $B_a\mu = \mathcal{S}_{a\mu}^{\mathbb{B}}$,
- $(U \Rightarrow V)\mu = U\mu \Rightarrow^* V\mu$.

► **Lemma 19.** *Assume that an annotation C for constructors is given.*

1. *Let c be an arbitrary constructor with $p^c \neq 0$, and ν, μ be arbitrary valuations from $\{\alpha_1, \dots, \alpha_{q^c}\}$ to \mathfrak{h} . If $\alpha_i \nu \leq \alpha_i \mu$ holds for any $i \in \{1, \dots, q^c\}$ with $\alpha_i \in \mathbb{V}$, then $\sigma^c \nu \leq \sigma^c \mu$ holds.*
2. *Let Σ^c be the size function c induced by C . If $\sigma^c \neq \infty$ holds, then $\mathbf{a}_i < \Sigma^c(\vec{\alpha})$ holds for any $\vec{\alpha}$ and any $i \in \{1, \dots, p^c\}$.*

Proof.

- (1.) Since $p^c \neq 0$ holds, we have $\text{Var}(\sigma^c) \subseteq \{\alpha_1, \dots, \alpha_{q^c}\}$. Then, the assertion follows because $\text{Pos}(\alpha_i^c, \sigma^c) \subseteq \text{Pos}^+(\alpha_i^c, \sigma^c)$ holds for any $i \in \{1, \dots, q^c\}$ by the definition of C .
- (2.) By the definition of C , if $\sigma \neq \infty$ holds then we have $\alpha_i^c <_{\mathbb{A}} \sigma^c$ for any $i \in \{1, \dots, p^c\}$. It follows from the definition of size algebras that $\alpha_i^c \nu < \sigma^c \nu$ holds where ν is the valuation defined in Definition 18, hence we have $\mathbf{a}_i < \Sigma^c(\vec{\alpha})$ for any $\vec{\alpha}$ and any $i \in \{1, \dots, p^c\}$. ◀

Without loss of generality, we may assume that for any $f \in \mathbb{F}$, there is a natural number $q^f \geq 0$ such that the first q^f arguments of f are all sorts. We denote the i -th argument of f by \mathbb{B}_i^f for any $i \in \{1, \dots, q^f\}$. Notice that there may be a natural number $i \in \{q^f + 1, \dots, r^f\}$ with T_i a sort. Using these notations, we define *annotations for function symbols* as follows:

► **Definition 20** (Annotations for Function Symbols). *Let \mathbb{A} be an arbitrary size algebra. An annotation F for function symbols consists of a well founded quasi ordering $\leq_{\mathbb{F}}$ on $\mathbb{F} \cup \mathbb{C} \cup \mathbb{V}$ and a family $((\mathbb{D}_{\mathbb{A}}^f, \leq_{\mathbb{A}}^f, <_{\mathbb{A}}^f, \zeta_{\mathbb{A}}^f), (\mathbb{D}_{\mathbb{h}}^f, \leq_{\mathbb{h}}^f, <_{\mathbb{h}}^f, \zeta_{\mathbb{h}}^f), \bar{\Theta}(f))_{f \in \mathbb{F}}$ which satisfy the following: for any $f \in \mathbb{F}$,*

1. $h <_{\mathbb{F}} f$ holds for any $h \in \mathbb{C} \cup \mathbb{V}$,
2. $\overline{\Theta}(f) = \overline{T}_1 \Rightarrow \dots \Rightarrow \overline{T}_{r^f} \Rightarrow B_{\sigma^f}$ with $\sigma^f \in \overline{A}$,
3. $\overline{T}_i = \text{Annot}(B_i^f, B_i^f, \alpha_i^f)$ for any $i \in \{1, \dots, q^f\}$, and $\overline{T}_i = T_i$ for any $i \in \{q^f + 1, \dots, r^f\}$,
4. $\text{Var}(\sigma^f) \subseteq \{\alpha_1^f, \dots, \alpha_{q^f}^f\} \subseteq \mathbb{V}$, and $\alpha_1^f, \dots, \alpha_{q^f}^f$ are pairwise distinct,
5. for each $X \in \{\mathbf{A}, \mathbf{h}\}$, \mathbb{D}_X^f is a set, \leq_X^f is a quasi ordering on \mathbb{D}_X^f , $<_X^f$ is a well founded relation with $<_X^f \subseteq \leq_X^f$, and ζ_X^f is a mapping from X^{q^f} to \mathbb{D}_X^f such that
 - if $f \simeq_{\mathbb{F}} g$ holds, then $(\mathbb{D}_X^f, \leq_X^f, <_X^f) = (\mathbb{D}_X^g, \leq_X^g, <_X^g)$ holds for each $X \in \{\mathbf{A}, \mathbf{h}\}$,
 - if $\vec{a} <_{\mathbf{A}}^{\mathbf{g}, \mathbf{f}} \vec{b}$ holds, then $\vec{a}\mu <_{\mathbf{h}}^{\mathbf{g}, \mathbf{f}} \vec{b}\mu$ for any valuation $\mu : \mathbb{V} \rightarrow \mathbf{h}$,
 - $<_{\mathbf{h}}^{\mathbf{g}, \mathbf{f}} \circ \leq_{\text{prod}} \subseteq <_{\mathbf{h}}^{\mathbf{g}, \mathbf{f}}$,
 where \circ denotes the composition of two relations, and for each $X \in \{\mathbf{A}, \mathbf{h}\}$,
 $(x_1, \dots, x_{q^g}) <_X^{\mathbf{g}, \mathbf{f}} (y_1, \dots, y_{q^f})$ holds iff $g \simeq_{\mathbb{F}} f$ and $\zeta_X^g(x_1, \dots, x_{q^g}) <_X^f \zeta_X^f(y_1, \dots, y_{q^f})$ hold.

As an example from [9, Example 3], we consider the subtraction $\text{sub} \in \mathbb{F}$ on natural numbers: we have $\Theta(\text{sub}) = \mathbb{N} \Rightarrow \mathbb{N} \Rightarrow \mathbb{N}$, and the rewrite rules are

$$\text{sub } x \ 0 \rightarrow x \qquad \text{sub } 0 \ y \rightarrow 0 \qquad \text{sub } (s \ x)(s \ y) \rightarrow \text{sub } x \ y$$

where $0 : \mathbb{N}$ (zero) and $s : \mathbb{N} \Rightarrow \mathbb{N}$ (the successor function) are the constructors for \mathbb{N} . Using the successor size algebra, we annotate $0, s$ and sub as follows:

$$\overline{\Theta}(0) = N_{\alpha}, \overline{\Theta}(s) = N_{\beta} \Rightarrow N_{s\beta}, \overline{\Theta}(\text{sub}) = N_{\gamma} \Rightarrow \mathbb{N} \Rightarrow N_{\gamma} \text{ with } q^{\text{sub}} = 1 \text{ and } \alpha^{\text{sub}} = \sigma^{\text{sub}} = \gamma.$$

The annotation $\overline{\Theta}(\text{sub})$ indicates that the size of $\text{sub } x \ y$ does not increase from the size of x . For each $X \in \{\mathbf{A}, \mathbf{h}\}$, we take $\mathbb{D}_X^{\text{sub}}$ as X , and $\zeta_X^{\text{sub}} : X \rightarrow \mathbb{D}_X^{\text{sub}}$ as the identity function. In addition, $\leq_{\mathbf{h}}^{\text{sub}}$ and $<_{\mathbf{h}}^{\text{sub}}$ are defined as \leq and $<$ on ordinals, respectively, while we put $\leq_{\mathbf{A}}^{\text{sub}} := \leq_{\mathbf{A}}$ and $<_{\mathbf{A}}^{\text{sub}} := <_{\mathbf{A}}$. The annotation with the identity function is often useful, as we will see below.

Notice that when both an annotation for constructors and an annotation of function symbols are given, we have a mapping $\overline{\Theta} : \mathbb{C} \cup \mathbb{F} \rightarrow \mathbb{T}_{\mathbf{A}}$. We adopt similar notations for $\overline{\Theta}$ to the ones for $\Theta : \mathbb{C} \cup \mathbb{F} \rightarrow \mathbb{T}$.

Below we define two orderings $<_{\mathbf{A}}$ and $<_{\mathbf{h}}$ on function calls, where a function call is expressed as a pair (f, ξ) of a function symbol f and a substitution ξ with the domain $\{\alpha_1^f, \dots, \alpha_{q^f}^f\}$. Here the substitution ξ provides an instance of function calls by instantiating the variables $\alpha_1^f, \dots, \alpha_{q^f}^f$.

► **Definition 21** (The Orderings on Function Calls). *Let $g, f \in \mathbb{F}$ be the case. Assume that an annotation for function symbols is given and that for each $X \in \{\mathbf{A}, \mathbf{h}\}$, two mappings $\xi : \{\alpha_1^g, \dots, \alpha_{q^g}^g\} \rightarrow X$ and $\eta : \{\alpha_1^f, \dots, \alpha_{q^f}^f\} \rightarrow X$ are given. Then, $(g, \xi) <_X (f, \eta)$ holds iff either $g <_{\mathbb{F}} f$ or $(\alpha_1^g \xi, \dots, \alpha_{q^g}^g \xi) <_X^{\mathbf{g}, \mathbf{f}} (\alpha_1^f \eta, \dots, \alpha_{q^f}^f \eta)$ holds.*

The following type system gives a condition of the termination criterion below (see the condition **Subject Reduction and Decreasingness**). Roughly speaking, the rule (app-decr) guarantees that the size of any function call in $ht^{\vec{t}}$ is strictly less than (f, φ) , and the rule (sub) with the subtyping rules enables us to reason about the subtyping relation between annotated types.

► **Definition 22** (Typing Rules for the Computability Closure). *Assume that annotations for \mathbb{C} and \mathbb{F} are given, and let f be a function symbol with a symbolic valuation $\varphi : \{\vec{\alpha}^f\} \rightarrow \mathbf{A}$. The typing rules for the computability closure of (f, φ) are as follows:*

Rule (app-decr). *If the following conditions*

1. $(h, \vec{V} \Rightarrow V) \in \Gamma \cup \bar{\Theta}$,
 2. either $h \simeq_{\mathbb{F}} f$ holds or $h <_{\mathbb{F}} f$ holds,
 3. either $h \in \mathbb{F}$ holds and ψ is a symbolic valuation from $\{\bar{\alpha}^h\} \rightarrow \mathbf{A}$ with $(h, \psi) <_{\mathbf{A}} (f, \varphi)$, or $h \in \mathbb{V} \cup \mathbb{C}$ holds and ψ is the empty function,
- are satisfied, then the rule

$$\frac{\Gamma \vdash_{\varphi}^f t_1 : V_1 \psi \quad \cdots \quad \Gamma \vdash_{\varphi}^f t_{|\vec{V}|} : V_{|\vec{V}|} \psi}{\Gamma \vdash_{\varphi}^f h \vec{t} : V \psi} \text{ (app-decr)}$$

is a typing rule for the computability closure of (f, φ) .

Rules (lam) and (sub). We have the following rules:

$$\frac{\Gamma, x : U \vdash_{\varphi}^f t : V}{\Gamma \vdash_{\varphi}^f \lambda x^U t : U \Rightarrow V} \text{ (lam)} \quad \frac{\Gamma \vdash_{\varphi}^f t : U \quad U \leq V}{\Gamma \vdash_{\varphi}^f t : V} \text{ (sub)}$$

where the subtyping rules are as follows:

$$\frac{a \leq_{\mathbf{A}}^{\infty} b}{\mathbf{B}_a \leq \mathbf{B}_b} \text{ (size)} \quad \frac{U' \leq U \quad V \leq V'}{U \Rightarrow V \leq U' \Rightarrow V'} \text{ (prod)} \quad \frac{}{U \leq U} \text{ (refl)} \quad \frac{U \leq V \quad V \leq T}{U \leq T} \text{ (tran)}$$

► **Remark 23.** In [9], the rule (app-decr) has one more condition which states that h is applied to at least q^h arguments whenever $h \in \mathbb{F}$ and $h \simeq_{\mathbb{F}} f$ holds. That is, the rule in [9] is obtained from the rule (app-decr) above by replacing the clause 2. with the clause 2'. either $h \simeq_{\mathbb{F}} f$ and $|\vec{V}| \geq q^h$ holds or $h <_{\mathbb{F}} f$ holds.

In fact, this condition is not needed to prove the correctness of the termination criterion in [9]. If this condition is dropped, then the criterion in [9] guarantees the termination of Hofmann's extract function for the breadth-first traversal of trees (see, e.g., [19]) as our termination criterion does. We will show in Example 27 below that our criterion guarantees the termination of Hofmann's extract function. If one keeps the condition 2', then it is, to the best of our knowledge, an open question whether the termination of Hofmann's extract function can be shown by the criterion in [9].

As an illustration of typing rules for the computability closure, we consider the annotated rewrite system of sub which we have seen above (see also [9, Example 3]). When the symbolic valuations ψ and φ are defined as $\psi := \{(\gamma, \beta)\}$ and $\varphi := \{(\gamma, \mathbf{s} \beta)\}$, respectively, then $(\text{sub}, \psi) <_{\mathbf{A}} (\text{sub}, \varphi)$ holds. Thus we have, e.g.,

$$\frac{\frac{x : \mathbf{N}_{\beta}, y : \mathbf{N} \vdash_{\varphi}^{\text{sub}} x : \mathbf{N}_{\beta} (= \mathbf{N}_{\gamma} \psi) \quad x : \mathbf{N}_{\beta}, y : \mathbf{N} \vdash_{\varphi}^{\text{sub}} y : \mathbf{N} (= \mathbf{N}_{\psi})}{x : \mathbf{N}_{\beta}, y : \mathbf{N} \vdash_{\varphi}^{\text{sub}} \text{sub } x y : \mathbf{N}_{\beta}} \text{ (app-decr)} \quad \frac{\beta \leq_{\mathbf{A}}^{\infty} \mathbf{s} \beta}{\mathbf{N}_{\beta} \leq \mathbf{N}_{\mathbf{s}\beta}} \text{ (size)}}{x : \mathbf{N}_{\beta}, y : \mathbf{N} \vdash_{\varphi}^{\text{sub}} \text{sub } x y : \mathbf{N}_{\mathbf{s}\beta}} \text{ (sub)}$$

When an annotation C for constructors and an annotation F for function symbols are given, we denote the set of all size variables used in C by $\mathbb{V}(C)$, and the set of all size variables used in F by $\mathbb{V}(F)$.

In sum, if one provides an annotation for constructors then this annotation determines a family of size functions, and this family determines a stratification as we have seen in Section 3. If one also gives an annotation for function symbols, one obtains the typing rules above for each function symbol f and symbolic valuation φ . Then, in order to prove the termination of a given rewrite relation \rightarrow , it suffices to show that our termination criterion below is satisfied.

The difference between our termination criterion below and the criterion of [9] consists in the condition (3) of **Accessibility** and the condition (2b) of **Minimality**. Both of these conditions are added in order to deal with constructors of non-positive types. In particular, the condition (3) of **Accessibility** includes the quasi-accessibility, which was defined above (Definition 14).

► **Theorem 24** (Termination Criterion). *Let C be an annotation for constructors and $(\Sigma^c)_{c \in \mathbb{C}}$ be the family of size functions induced by C such that for any c and any \vec{a} , $\Sigma^c(\vec{a})$ is not a limit ordinal. Moreover, let an annotation F for function symbols be given. Then, the rewrite relation \rightarrow terminates on the set of all well-typed terms if for each rule $f l_1 \cdots l_{|\vec{l}|} \rightarrow r \in \mathcal{R}$ with $\vec{\Theta}(f) = \vec{T}_1 \Rightarrow \cdots \Rightarrow \vec{T}_{r^f} \Rightarrow \mathbf{B}_{\sigma^f}$,*

- $|\vec{l}| \geq q^f$ holds,
- there is a typing environment $\Gamma : \text{FV}(r) \rightarrow \mathbb{T}_A$ satisfying the following: for any $(x, U) \in \Gamma$, there are a term l_k ($1 \leq k \leq |\vec{l}|$), a sort \mathbf{B}^x and a size variable $\alpha^x \in \mathbb{V}(C) \cup \mathbb{V}(F)$ with $\text{Pos}(x, l_k) \neq \emptyset$ and $U = \text{Annot}(|U|, \mathbf{B}^x, \alpha^x)$,
- there is a substitution $\varphi : \{\alpha_1, \dots, \alpha_{q^f}\} \rightarrow A$ such that the following conditions are satisfied:

Monotony. For any $i \in \{1, \dots, q^f\}$, $\text{Pos}(\alpha_i^f, \sigma^f) \subseteq \text{Pos}^+(\alpha_i^f, \sigma^f)$ holds.

Accessibility. For any $(x, U) \in \Gamma$,

1. $x = l_k$ and $U = \vec{T}_k \varphi$ holds, or
2. $\text{Pos}(\mathbf{B}^x, |U|) \subseteq \text{Pos}^+(\mathbf{B}^x, |U|)$ holds, T_k is a sort and $(x, |U|, \mathbf{B}^x) \leq_a (l_k, T_k, T_k)$, or
3. $\text{Pos}(\mathbf{B}^x, |U|) \not\subseteq \text{Pos}^+(\mathbf{B}^x, |U|)$ holds, T_k is a sort and $(x, |U|, \mathbf{B}^x) \leq_{\text{qa}} (l_k, T_k, T_k)$.

Minimality. For any substitution θ such that $l_j \theta \in T_j$ holds for any $j \in \{1, \dots, |\vec{l}|\}$, there is an ordinal valuation ν satisfying

1. for any $i \in \{1, \dots, q^f\}$, $\alpha_i^f \varphi \nu = o_{\mathcal{S}^{\mathbf{B}_i}}(l_i \theta)$ holds, and
2. for any $(x, U) \in \Gamma$,
 - a. if $\text{Pos}(\mathbf{B}^x, |U|) \subseteq \text{Pos}^+(\mathbf{B}^x, |U|)$ holds, then $o_{[\mathbf{B}^x : \mathcal{S}^{\mathbf{B}^x}] |U|}(x \theta) \leq \alpha^x \nu$ holds,
 - b. otherwise $o_{(\bigcup_{c \leq a} [\mathbf{B}^x : \mathcal{S}_c^{\mathbf{B}^x}] |U|)_{a < b}}(x \theta) = \alpha^x \nu$ holds.

Subject Reduction and Decreasingness. $\Gamma \vdash_{\varphi}^f r : T_{|\vec{l}|+1} \Rightarrow \cdots \Rightarrow T_{r^f} \Rightarrow \mathbf{B}_{\sigma^f} \varphi$ holds.

Proof. Below we show the computability of constructors, the computability of function symbols and the computability of well-typed terms one by one. We first prove the computability of constructors: for any $c \in \mathbb{C}$ with $\vec{\Theta}(c) = \vec{T}_1 \Rightarrow \cdots \Rightarrow \vec{T}_{r^c} \Rightarrow \mathbf{B}_{\sigma^c}$, any $\mu : \mathbb{V} \rightarrow \mathfrak{h}$ and any \vec{t} with $|\vec{t}| = r^c$, if $t_i \in \vec{T}_i \mu$ holds for any i with $1 \leq i \leq r^c$, then $c \vec{t} \in \mathbf{B}_{\sigma^c} \mu$ holds. Here we need to consider constructors of non-positive types, which were not handled by [9]. Let $c \in \mathbb{C}$ be the case with $\vec{\Theta}(c) = \vec{T}_1 \Rightarrow \cdots \Rightarrow \vec{T}_{r^c} \Rightarrow \mathbf{B}_{\sigma^c}$. In addition, let $\mu : \mathbb{V} \rightarrow \mathfrak{h}$ be an arbitrary valuation, and assume that $t_i \in \vec{T}_i \mu$ holds for any i with $1 \leq i \leq r^c$. Putting $\vec{t} := t_1, \dots, t_{r^c}$, we verify $c \vec{t} \in \mathbf{B}_{\sigma^c} \mu$. It is obvious that $c \vec{t} \in \text{SN}$ holds. If $p^c = 0$ holds then we have $c \vec{t} \in \mathcal{S}_0^{\mathbf{B}}$ by Definition 10, hence $c \vec{t} \in \mathbf{B}_{\sigma^c} \mu$ follows. Below we assume that $p^c \neq 0$ holds.

Consider an arbitrary $(c, \vec{u}, \vec{T}) \in \mathbb{C}_{\rightarrow}^{\mathbf{B}}(c \vec{t})$. If $i \in \{1, \dots, n^c\}$ holds, then we have $u_i \in \vec{T}_i \mu = [\mathbf{B}_{\alpha_i} : \mathcal{S}_{\alpha_i \mu}^{\mathbf{B}}] \vec{T}_i \mu \subseteq \bigcup_{c \leq \alpha_i \mu} [\mathbf{B} : \mathcal{S}_c^{\mathbf{B}}] T_i$, because $t_i \rightarrow^* u_i$ holds and $\vec{T}_i \mu$ is a computability predicate. Similarly, for each $i \in \{n^c + 1, \dots, q^c\}$, we have $u_i \in \vec{T}_i \mu = [\mathbf{B}_i : \mathcal{S}_{\alpha_i \mu}^{\mathbf{B}_i}] T_i$. Put \mathbf{a} as $\mathbf{a} := \max(\{\Sigma^c(o_{\mathcal{S}^{c,1}}(u_1), \dots, o_{\mathcal{S}^{c,q^c}}(u_{q^c}))\} \cup \{\alpha_i \mu \mid 1 \leq i \leq q^c, \alpha_i \in \mathbb{V}\})$. Then, $u_i \in \bigcup_{c \leq \mathbf{a}} [\mathbf{B} : \mathcal{S}_c^{\mathbf{B}}] T_i$ holds for any $i \in \{1, \dots, n^c\}$, and $u_i \in [\mathbf{B}_i : \mathcal{S}_{\mathbf{a}}^{\mathbf{B}_i}] T_i$ holds for any $i \in \{n^c + 1, \dots, q^c\}$ by Proposition 3. Therefore, $c \vec{t} \in \mathcal{S}_{\mathbf{a}+1}^{\mathbf{B}}$ holds and $o_{\mathcal{S}^{\mathbf{B}}}(c \vec{t})$ exists.

If $\sigma^c = \infty$ holds then we immediately have $c \vec{t} \in \mathbf{B}_{o_{\mathcal{S}^{\mathbf{B}}}(c \vec{t})} \subseteq \mathbf{B}_{\sigma^c \mu}$, so let $\sigma^c \neq \infty$ be the case. By Lemma 13.(2), we have $o_{\mathcal{S}^{\mathbf{B}}}(c \vec{t}) = \Sigma(o_{\mathcal{S}}(\vec{t})) = \sigma^c \nu$, where ν is the valuation defined in Definition 18. If $\alpha_i \nu \leq \alpha_i \mu$ holds for any $\alpha_i \in \mathbb{V} \cap \{\vec{\alpha}\}$, then it follows from $\text{Var}(\sigma^c) \subseteq \{\vec{\alpha}\}$ and Lemma 19.(1) that $\sigma^c \nu \leq \sigma^c \mu$ holds. We show that $\alpha_i \nu \leq \alpha_i \mu$ holds for any $\alpha_i \in \mathbb{V} \cap \{\vec{\alpha}\}$.

Since we consider the case of $\sigma^c \neq \infty$, $\{\vec{\alpha}\} \cap \mathbf{V}$ must be non-empty. If the members of $\{\vec{\alpha}\} \cap \mathbf{V}$ are pairwise distinct, we have $\alpha_i \nu = o_{\mathcal{S}^i}(t_i) \leq \alpha_i \mu$ because $t_i \in \overline{T}_i \mu$ holds. Assume that all members of $\{\vec{\alpha}\} \cap \mathbf{V}$ are equal. We have

$$\alpha_i \nu = \sup\{o_{\mathcal{S}^j}(t_j) \mid 1 \leq j \leq q^c, \alpha_j \in \mathbf{V}\} = o_{\mathcal{S}^k}(t_k)$$

for some k with $1 \leq k \leq q^c$ by definition. Therefore, $\alpha_i \nu = \alpha_k \nu \leq \alpha_k \mu = \alpha_i \mu$ holds because we have $t_k \in \overline{T}_k \mu$.

Next, we show the computability of function symbols: we verify that for any $f \in \mathbb{F}$ with $\overline{\Theta}(f) = \overline{T}_1 \Rightarrow \cdots \Rightarrow \overline{T}_{r^f} \Rightarrow \mathbf{B}_{\sigma^f}$, any $\mu : \{\alpha_1^f, \dots, \alpha_{q^f}^f\} \rightarrow \mathfrak{h}$ and any \vec{t} with $|\vec{t}| = r^f$, if $t_i \in \overline{T}_i \mu$ holds for any i , then $f \vec{t} \in \mathbf{B}_{\sigma^f} \mu$ holds. We show this claim by induction on $((f, \mu), \vec{t})$ with $(\langle \cdot \rangle_{\mathfrak{h}}, \leftarrow_{\text{prod}})_{\text{lex}}$. Let $t_i \in \overline{T}_i \mu$ be the case for any i with $1 \leq i \leq r^f$. Since $f \vec{t}$ is neutral, it suffices to show that $u \in \mathbf{B}_{\sigma^f} \mu$ holds for any u with $f \vec{t} \rightarrow u$. The case of $u = f \vec{u}$ and $\vec{t} \rightarrow_{\text{prod}} \vec{u}$ is straightforward. Otherwise, we have

(*) $f \vec{t} \rightarrow r \in \mathcal{R}$, $\vec{t} = \vec{l} \theta \vec{u}$ and $u = r \theta \vec{u}$,

where $\vec{u} = t_{|\vec{l}|+1}, \dots, t_{r^f}$ holds. Recall that if $i \leq q^f$ holds then T_i is a sort and that if $i > q^f$ holds then $\overline{T}_i = T_i$ holds. Therefore, for any $i \in \{1, \dots, |\vec{l}|\}$, we have $l_i \theta \in T_i$ by $l_i \theta \in \overline{T}_i \mu$. We obtain the following valuation ν by Minimality: for any $i \in \{1, \dots, q^f\}$, $\alpha_i^f \varphi \nu = o_{\mathcal{S}^i}(l_i \theta)$ holds, and for any $(x, U) \in \Gamma$,

1. if $\text{Pos}(\mathbf{B}^x, |U|) \subseteq \text{Pos}^+(\mathbf{B}^x, |U|)$ holds, then $o_{[\mathbf{B}^x, \mathcal{S}^{\mathbf{B}^x}]|U|}(x\theta) \leq \alpha^x \nu$ holds,
2. otherwise $o_{(\bigcup_{c \leq a} [\mathbf{B}^x, \mathcal{S}^{\mathbf{B}^x}]|U|)_{a < b}}(x\theta) = \alpha^x \nu$ holds.

We prove the following claims 1–3. The claim 3 will be proved in the same way as [9, Theorem 1]. On the other hand, the claim 1 is proved without a condition on (app-decr) which was imposed in [9] (see Remark 23 above), and the claim 2 forces us to consider the new case where \mathbf{B}^x occurs in $|U|$ negatively for some $(x, U) \in \Gamma$.

1. **Correctness of the computability closure:** let ν be the valuation given above by Minimality, and φ be the substitution given by the assumptions of this theorem. Using subinduction on \vdash_{φ}^f , we show that for any typing environment Δ , any term t , any type T and any substitution θ' , if (i) $\Delta \vdash_{\varphi}^f t : T$ holds and (ii) $x\theta' \in U\nu$ holds for any $(x, U) \in \Delta$, then $t\theta' \in T\nu$ holds.

We consider the principal case (app-decr) only: let $\Delta \vdash_{\varphi}^f h\vec{w} : V\psi$ be the case, then we have $w_i \theta' \in V_i \psi \nu$ for any i with $1 \leq i \leq |\vec{V}|$ by the hypothesis of subinduction. We put $k := |\vec{V}|$. If $h \in \mathbb{V}$ holds, then $\psi = \emptyset$ holds and we have $h\theta' \in (\vec{V} \Rightarrow V)\nu$ by assumption, hence we have $h\theta' \vec{w}\theta' \in V\psi \nu$. Let $h \in \mathbb{C}$ be the case, and put $V = (U_1 \Rightarrow \cdots \Rightarrow U_m \Rightarrow \mathbf{C}_{\sigma^h})$. To show $h\vec{w}\theta' \in V\psi \nu$, consider arbitrary $u_1 \in U_1 \psi \nu, \dots, u_m \in U_m \psi \nu$. By the computability of constructors, we have $h\vec{w}\theta' \vec{u} \in \mathbf{C}_{\sigma^h} \psi \nu$, hence $h\vec{w}\theta' \in V\psi \nu$ holds by definition. Let $h \in \mathbb{F}$ be the case, and assume that

$$V = (U_1 \Rightarrow \cdots \Rightarrow U_m \Rightarrow \mathbf{C}_{\sigma^h}), \quad \vec{\alpha}^h = \alpha_1^h, \dots, \alpha_{q^h}^h, \quad \vec{\alpha}^f = \alpha_1^f, \dots, \alpha_{q^f}^f$$

holds. If $h \in \mathbb{F}$ holds, then $(h, \psi \nu) <_{\mathfrak{h}} (f, \mu)$ immediately follows from Definition 21. Otherwise, we have $\vec{\alpha}^h \psi <_{\mathbf{A}}^{h, f} \vec{\alpha}^f \varphi$ by $(h, \psi) <_{\mathbf{A}} (f, \varphi)$. Then, by Definition 20, we have $\vec{\alpha}^h \psi \nu <_{\mathfrak{h}}^{h, f} \vec{\alpha}^f \varphi \nu$. By the minimality of ν , we have $\vec{\alpha}^f \varphi \nu \leq_{\text{prod}} \vec{\alpha}^f \mu$, so we have $\vec{\alpha}^h \psi \nu <_{\mathfrak{h}}^{h, f} \vec{\alpha}^f \mu$ by Definition 20 again. Therefore, $(h, \psi \nu) <_{\mathfrak{h}} (f, \mu)$ holds also in this case. By the hypothesis of the main induction, for any \vec{t} with $|\vec{t}| = t_1, \dots, t_k, t_{k+1}, \dots, t_{k+m}$, if $t_i \in V_i \psi \nu$ holds for any i with $1 \leq i \leq k$ and $t_{k+j} \in U_j \psi \nu$ holds for any j with $1 \leq j \leq m$, then $h\vec{t} \in \mathbf{C}_{\sigma^h} \psi \nu$ holds. Therefore, we have $h\vec{w}\theta' \vec{u} \in \mathbf{C}_{\sigma^h} \psi \nu$ for any $u_1 \in U_1 \psi \nu, \dots, u_m \in U_m \psi \nu$, hence $h\vec{w}\theta' \in V\psi \nu$ holds by definition.

2. **Computability of the matching substitution:** we show that $x\theta \in U\nu$ holds for any $(x, U) \in \Gamma$, where θ is the substitution given in (*) and $\Gamma : \text{FV}(r) \rightarrow \mathbb{T}_A$ is the typing environment given by the assumptions of the theorem. Consider an arbitrary $(x, U) \in \Gamma$, then there is an integer k , a term l_k , a sort \mathbb{B}^x and a size variable α^x such that all of them satisfy the assumptions of the theorem. If $\text{Pos}(\mathbb{B}^x, |U|) \subseteq \text{Pos}^+(\mathbb{B}^x, |U|)$ holds then we can prove the assertion as in [9, Theorem 1] by using Lemma 15 and Accessibility, so assume that \mathbb{B}^x occurs in $|U|$ negatively. If $x = l_k$ holds, then k must be greater than q^f because $|U|$ is not a sort, hence we have

$$|U| = U = T_k = \overline{T_k}$$

and so $x\theta = l_k\theta \in \overline{T_k}\mu = U\nu$ holds. Let $x \neq l_k$ be the case, and suppose that the position of $x\theta$ in $l_k\theta$ is $p_1 \cdots p_i$ ($i \geq 1$). By Accessibility, we have $(x, |U|, \mathbb{B}^x) \triangleleft_{\text{qa}} (l_k, T_k, T_k)$. Therefore, for any subterm t of $l_k\theta$ whose position in $l_k\theta$ is ϵ or $p_1 \cdots p_j$ for some j with $0 \leq j < i$, t has the form $c \vec{w}$ for some $c \in \mathbb{C}$ and some \vec{w} . Let $c_0 \vec{w}_0, c_1 \vec{w}_1, \dots, c_{i-1} \vec{w}_{i-1}$ be the subterms of positions $\epsilon, p_1, \dots, p_1 \cdots p_{i-1}$ in $l_k\theta$, respectively (hence $c_0 \vec{w}_0 = l_k\theta$). We put $c := c_{i-1}$ if $i - 1 \neq 0$, otherwise we put $c := c_0$. Then, since $l_k\theta \in \overline{T_k}\mu$ holds, we have $x\theta \in \bigcup_{c \leq a} [\mathbb{B}^c : \mathcal{S}_c^{\mathbb{B}^c}]|U|$ for some a by applying Definition 10 repeatedly. By the minimality of ν , we have $x\theta \in U\nu$.

3. We show that $u \in \mathbb{B}_{\sigma^f}\mu$ holds. The claims 1. and 2. above show $r\theta \in \overline{T_{|l|+1}}\varphi\nu \Rightarrow \dots \Rightarrow \overline{T_{r^f}}\varphi\nu \Rightarrow \mathbb{B}_{\sigma^f}\varphi\nu$. By definition, we have $\overline{T_i} = T_i$ for any i with $|l| + 1 \leq i \leq r^f$, so we have $r\theta \vec{u} \in \mathbb{B}_{\sigma^f}\varphi\nu$. If $\sigma^f = \infty$ holds then we immediately have $\mathbb{B}_{\sigma^f}\varphi\nu = \mathbb{B}_{\sigma^f}\mu$, so let $\sigma^f \neq \infty$ be the case. Since φ does not assign ∞ to any of $\alpha_1, \dots, \alpha_{q^f}$, we have $\sigma^f\varphi \neq \infty$, so $\nu(\sigma^f\varphi)$ is defined. By Monotony, it suffices to verify $\alpha_i\varphi\nu \leq \alpha_i\mu$ for any i with $1 \leq i \leq q^f$, but this follows from Minimality.

Finally, following [9, Theorem 1], we show the computability of well-typed terms by induction on \vdash : we show that for any Γ, t, T with $\Gamma \vdash t : T$ and any substitution θ , if $x\theta \in U$ holds for any $(x, U) \in \Gamma$, then $t\theta \in T$ holds. The termination of \rightarrow follows from the computability of well-typed terms: if $\Gamma \vdash t : T$ holds, then we consider the empty substitution θ . Since any computability predicate subsumes the set \mathbb{V} of variables, we have $x\theta = x \in U$ for any $(x, U) \in \Gamma$, hence $t \in T \subseteq \text{SN}$ holds.

The case of function symbols: by assumption, we have $\vdash f : \Theta(f)$ with $\Theta(f) = T_1 \Rightarrow \dots \Rightarrow T_{r^f} \Rightarrow \mathbb{B}$ and $\overline{\Theta}(f) = \overline{T_1} \Rightarrow \dots \Rightarrow \overline{T_{r^f}} \Rightarrow \mathbb{B}_{\sigma^f}$. It suffices to verify that $f \vec{t} \in \mathbb{B}$ for any $\vec{t} \in \vec{T}$ with $|\vec{t}| = r^f$. Define the valuation $\mu : \{\alpha_1, \dots, \alpha_{q^f}\} \rightarrow \mathfrak{h}$ as $\mu(\alpha_i) := \mathfrak{m}(\mathcal{S}^{\mathbb{B}^i})$. Since $T_i = \mathbb{B}_i$ holds for any $i \in \{1, \dots, q^f\}$ and $\overline{T_j} = T_j$ holds for any $j \in \{q^f + 1, \dots, r^f\}$, we have $t_k \in \overline{T_k}\mu$ for any $k \in \{1, \dots, r^f\}$. By the computability of function symbols, we have $f \vec{t} \in \mathbb{B}_{\sigma^f}\mu \subseteq \mathbb{B}$.

The case of constructors: by assumption, we have $\vdash c : \Theta(c)$ with $\Theta(c) = T_1 \Rightarrow \dots \Rightarrow T_{r^c} \Rightarrow \mathbb{B}$ and $\overline{\Theta}(c) = \overline{T_1} \Rightarrow \dots \Rightarrow \overline{T_{r^c}} \Rightarrow \mathbb{B}_{\sigma^c}$. It suffices to verify that $c \vec{t} \in \mathbb{B}$ for any $\vec{t} \in \vec{T}$ with $|\vec{t}| = r^c$. Define the valuation $\mu : \{\alpha_1, \dots, \alpha_{q^c}\} \rightarrow \mathfrak{h}$ as in the previous paragraph. Let $i \in \{1, \dots, q^c\}$ be the case. We have

$$t_i \in T_i = [\mathbb{B}_i : \mathbb{B}_i]T_i = [\mathbb{B}_i : \mathcal{S}_{\mu(\alpha_i)}^{\mathbb{B}_i}]T_i,$$

hence $t_i \in \overline{T_i}\mu$ holds. Therefore, we have $t_k \in \overline{T_k}\mu$ for any $k \in \{1, \dots, r^c\}$. By the computability of constructors, we have $c \vec{t} \in \mathbb{B}_{\sigma^c}\mu \subseteq \mathbb{B}$. The proofs of the other cases are standard. \blacktriangleleft

Below we discuss several examples of rewrite systems. The first and the second (Examples 25 and 26) are examples of non-terminating rewrite systems with a rule for non-positive types. The third and the fourth (Examples 27 and 28) contain a non-strictly positive inductive type and a non-positive type, respectively. We show that the first two examples cannot satisfy our termination criterion, and that the last two examples satisfy the criterion.

► **Example 25.** We consider the following non-terminating rewrite system with a non-positive type \mathbf{B} , which was discussed in [7]: put $\mathbb{S} := \{\mathbf{B}, \mathbf{A}\}$, $\mathbb{C} := \{\mathbf{c}\}$ and $\mathbb{F} := \{\mathbf{p}\}$ with $\Theta(\mathbf{c}) = (\mathbf{B} \Rightarrow \mathbf{A}) \Rightarrow \mathbf{B}$ and $\Theta(\mathbf{p}) = \mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{A}$. Define the rewrite system \mathcal{R} as $\mathcal{R} := \{\mathbf{p}(\mathbf{c}x) \rightarrow x\}$ with $x \in \mathbb{V}$. If we put $\omega := \lambda y^{\mathbf{B}} \mathbf{p} y y$, then we have

$$\omega(\mathbf{c} \omega) \rightarrow_{\beta} \mathbf{p}(\mathbf{c} \omega)(\mathbf{c} \omega) \rightarrow_{\mathcal{R}} \omega(\mathbf{c} \omega) \rightarrow_{\beta} \dots$$

so this rewrite system \mathcal{R} is non-terminating. The system \mathcal{R} cannot satisfy our termination criterion: suppose that \mathcal{R} satisfies it. Then, $q^{\mathbf{p}} \leq 1$ holds and so we have $\Gamma \vdash_{\varphi}^{\mathbf{p}} x : \mathbf{B} \Rightarrow \mathbf{A}_{\sigma} \varphi$ by Subject Reduction and Decreasingness. That is, $\mathbf{B}^x \neq \mathbf{B}$ holds and \mathbf{B} is not annotated in U with $(x, U) \in \Gamma$ and $U = \mathbf{B} \Rightarrow \mathbf{A}_{\sigma} \varphi$. Since $x \neq l_1$ holds, either $(x, |U|, \mathbf{B}^x) \triangleleft_a (l_1, T_1, T_1)$ or $(x, |U|, \mathbf{B}^x) \triangleleft_{\text{qa}} (l_1, T_1, T_1)$ must hold by Accessibility. Note that $T_1 = \mathbf{B}$ and $l_1 = \mathbf{c}x$ holds. If $(x, |U|, \mathbf{B}^x) \triangleleft_a (l_1, T_1, T_1)$ holds, this contradicts the fact that x is the first argument of \mathbf{c} and $n^{\mathbf{c}} = 1$ holds. If $(x, |U|, \mathbf{B}^x) \triangleleft_{\text{qa}} (l_1, T_1, T_1)$ holds, then we must choose \mathbf{B} in U as \mathbf{B}^x , but this contradicts the fact that \mathbf{B} in $U = \mathbf{B} \Rightarrow \mathbf{A}_{\sigma} \varphi$ is not annotated.

► **Example 26 (Higher-Order Abstract Syntax for Untyped λ -Calculus).** The following is a non-terminating rewrite system which is obtained from Example 25 above with a minor change: put $\mathbb{S} := \{\mathbf{B}\}$, $\mathbb{C} := \{\mathbf{abs}\}$ and $\mathbb{F} := \{\mathbf{app}\}$ with $\Theta(\mathbf{abs}) = (\mathbf{B} \Rightarrow \mathbf{B}) \Rightarrow \mathbf{B}$ and $\Theta(\mathbf{app}) = \mathbf{B} \Rightarrow \mathbf{B} \Rightarrow \mathbf{B}$. Then, define the rewrite system \mathcal{R} as $\mathcal{R} := \{\mathbf{app}(\mathbf{abs} g) x \rightarrow g x\}$ with $\{g, x\} \subseteq \mathbb{V}$. If we put $\omega' := \mathbf{abs}(\lambda y^{\mathbf{B}} \mathbf{app} y y)$, then

$$\mathbf{app} \omega' \omega' = \mathbf{app}(\mathbf{abs}(\lambda y^{\mathbf{B}} \mathbf{app} y y)) \omega' \rightarrow_{\mathcal{R}} (\lambda y^{\mathbf{B}} \mathbf{app} y y) \omega' \rightarrow_{\beta} \mathbf{app} \omega' \omega' \rightarrow_{\mathcal{R}} \dots$$

holds, hence this rewrite system \mathcal{R} is non-terminating.

We show that the system \mathcal{R} cannot satisfy our termination criterion. Assume that \mathcal{R} satisfies the criterion. By Accessibility, we have $\mathbf{B}^g = \mathbf{B}$ as in Example 25 above, so \mathbf{B} must be annotated in U with $(g, U) \in \Gamma$, that is, $(g, \mathbf{B}_{\alpha^g} \Rightarrow \mathbf{B}_{\alpha^g}) \in \Gamma$ holds. Then, $(x, \mathbf{B}_{\alpha^x}) \in \Gamma$ and $\alpha^x = \alpha^g$ must hold, because we have $\vdash_{\varphi}^{\mathbf{app}} g x : \mathbf{B}_{\sigma} \varphi$ by Subject Reduction and Decreasingness.

By Lemma 11.(3), $\mathbf{B}_0 \Rightarrow \mathbf{B}_0$ is a computability predicate, hence there exists a variable $z \in \mathbf{B}_0 \Rightarrow \mathbf{B}_0$ because any variable belongs to $\mathbf{B}_0 \Rightarrow \mathbf{B}_0$. This implies that $\mathbf{abs} z \in \mathbf{B}$ holds. We define a substitution θ as $\theta := \{(g, z), (x, \mathbf{abs} z)\}$, then $l_1 \theta = \mathbf{abs} g \theta \in \mathbf{B}$ and $l_2 \theta = x \theta \in \mathbf{B}$ holds. By Minimality, we have a valuation ν such that

$$o(\bigcup_{c \leq a} \mathbf{B}_{c \Rightarrow \mathbf{B}_c})_{a < b}(z) = o(\bigcup_{c \leq a} \mathbf{B}_{c \Rightarrow \mathbf{B}_c})_{a < b}(g \theta) = \alpha^g \nu = \alpha^x \nu \geq o_{\mathbf{B}}(x \theta) = o_{\mathbf{B}}(\mathbf{abs} z)$$

holds with $o(\bigcup_{c \leq a} \mathbf{B}_{c \Rightarrow \mathbf{B}_c})_{a < b}(z) = 0$, but this contradicts the definition of the stratification $\mathcal{S}^{\mathbf{B}}$.

In the two examples below, we use the successor size algebra, which was defined above.

► **Example 27 (Hofmann's Extract Function for the Breadth-First Traversal of Trees (see, e.g., [19])).** Let $\mathbb{S} := \{\mathbf{C}, \mathbf{L}\}$ be the set of sorts, $\mathbb{F} := \{\mathbf{e}\}$ be the set of function symbols with $\Theta(\mathbf{e}) = \mathbf{C} \Rightarrow \mathbf{L}$, and $\mathbb{C} := \{\mathbf{c}\}$ be the set of constructors with $\Theta(\mathbf{c}) = ((\mathbf{C} \Rightarrow \mathbf{L}) \Rightarrow \mathbf{L}) \Rightarrow \mathbf{C}$. The following rewrite rule satisfies our termination criterion: $\mathbf{e}(\mathbf{c}x) \rightarrow x \mathbf{e}$ with $x \in \mathbb{V}$.

We annotate c as $\bar{\Theta}(c) = ((C_\alpha \Rightarrow L) \Rightarrow L) \Rightarrow C_{s\alpha}$, and e as $\bar{\Theta}(e) = C_\beta \Rightarrow L_\infty$. By Definition 18, we have the size function Σ^c induced by this annotation of c . We define $\Gamma := \{x : (C_\alpha \Rightarrow L) \Rightarrow L\}$, $B^x := C$, $\alpha^x := \alpha$ and $\varphi := \{(\beta, s\alpha)\}$. Then, take ζ_X^e as the identity function for each $X \in \{A, h\}$.

Monotony. Obvious.

Accessibility. We have $(x, (C \Rightarrow L) \Rightarrow L, C) \leq_a (c\ x, C, C)$.

Minimality. Let θ be a substitution with $c\ x\theta \in C$. We define ν as $\nu := \{(\alpha, a)\}$ with $a := o_{[C; S^c](C \Rightarrow L) \Rightarrow L}(x\theta)$. Then, by Lemma 13, we have $\beta\varphi\nu = (s\alpha)\nu = a + 1 = \Sigma^c(a) = o_{S^c}(c\ x\theta)$.

Subject Reduction and Decreasingness. Put $\psi := \{(\beta, \alpha)\}$, then we have the following derivation:

$$\frac{x : (C_\alpha \Rightarrow L) \Rightarrow L \vdash_\varphi^e x : (C_\alpha \Rightarrow L) \Rightarrow L \quad \overline{\vdash_\varphi^e e : C_\alpha \Rightarrow L} \text{ (app-decr)}}{x : (C_\alpha \Rightarrow L) \Rightarrow L \vdash_\varphi^e x\ e : L} \text{ (app-decr)}$$

where $C_\alpha = C_\beta\psi$ and $(e, \psi) <_A (e, \varphi)$ hold. Note that we have $\vdash_\varphi^e e : C_\alpha \Rightarrow L$ because we dropped the condition $|\vec{V}| \geq q^h$ in the rule (app-decr) of [9] (see Remark 23 above).

► **Example 28** (β -Reduction and $\beta\eta$ -Reduction of Untyped λ -Calculus). Let $\mathbb{S} := \{B\}$ be the set of sorts, where B denotes the type of untyped λ -terms. The following rewrite rules say that abs_β is the λ -abstraction for β -reduction and that $\text{abs}_{\beta\eta}$ is the λ -abstraction for $\beta\eta$ -reduction:

$$\text{app}(\text{abs}_\beta g)\ x \rightarrow g\ x \quad \text{app}(\text{abs}_{\beta\eta} g)\ x \rightarrow g\ x \quad \text{abs}_{\beta\eta}(\lambda y^B \text{app}\ x\ y) \rightarrow x$$

where abs_β is a constructor of B , while $\text{abs}_{\beta\eta}$ and app are function symbols. In this way, one can deal with both of β -reduction and $\beta\eta$ -reduction in one rewrite system. Of course, this rewrite system is not terminating as seen above. On the other hand, consider the set \mathcal{R} of the following rewrite rules:

$$f(\text{abs}_\beta g) \rightarrow \text{abs}_{\beta\eta} g \quad f(\text{abs}_{\beta\eta} g) \rightarrow \text{abs}_\beta g \quad f(\text{app}\ x\ y) \rightarrow \text{app}(f\ x)(f\ y)$$

with $\{g, x, y\} \subseteq \mathbb{V}$. These rules enable us to replace the outermost λ -abstraction for β -reduction with the one for $\beta\eta$ -reduction, and vice versa. We show that the system \mathcal{R} satisfies the termination criterion. Note that abs_β , $\text{abs}_{\beta\eta}$ and app are all constructors of B when we consider the system \mathcal{R} . We thus assume that C is given as $\{\text{abs}_\beta, \text{abs}_{\beta\eta}, \text{app}\}$ with $\bar{\Theta}(\text{abs}_\beta) = (B_\gamma \Rightarrow B_\gamma) \Rightarrow B_{s\gamma}$, $\bar{\Theta}(\text{abs}_{\beta\eta}) = (B_{\gamma'} \Rightarrow B_{\gamma'}) \Rightarrow B_{s\gamma'}$ and $\bar{\Theta}(\text{app}) = B_{\gamma''} \Rightarrow B_{\gamma''} \Rightarrow B_{s\gamma''}$. By Definition 18, we have the size functions $\Sigma^{\text{abs}_\beta}$, $\Sigma^{\text{abs}_{\beta\eta}}$ and Σ^{app} . Moreover, put $\mathbb{F} := \{f\}$ with $\bar{\Theta}(f) = B_{\alpha^f} \Rightarrow B_{\alpha^f}$, and take ζ_X^f as the identity function for each $X \in \{A, h\}$.

Consider the first rule (the second rule can be handled in the same way). We define $\Gamma := \{g : B_{\gamma'} \Rightarrow B_{\gamma'}\}$, $B^g := B$, $\alpha^g := \gamma'$ and $\varphi := \{(\alpha^f, s\gamma')\}$.

Monotony. Obvious.

Accessibility. We have $(g, B \Rightarrow B, B) \leq_{qa} (\text{abs}_\beta g, B, B)$.

Minimality. Let θ be a substitution with $\text{abs}_\beta g\theta \in B$. We define ν as $\nu := \{(\gamma', a)\}$ with $a := o_{(\bigcup_{c \leq b} [B; S^c] B \Rightarrow B)_{b < h}}(g\theta)$. Then, by Lemma 13, we have $\alpha^f\varphi\nu = (s\gamma')\nu = a + 1 = \Sigma^{\text{abs}_\beta}(a) = o_{S^B}(\text{abs}_\beta g\theta)$.

Subject Reduction and Decreasingness. We have

$$\frac{\begin{array}{c} g : B_{\gamma'} \Rightarrow B_{\gamma'} \vdash_\varphi^f g : B_{\gamma'} \Rightarrow B_{\gamma'} \\ \vdots \\ g : B_{\gamma'} \Rightarrow B_{\gamma'} \vdash_\varphi^f \text{abs}_{\beta\eta} g : B_{s\gamma'} \end{array} \text{ (app-decr)} \quad \begin{array}{c} \vdots \\ B_{s\gamma'} \leq B \end{array}}{g : B_{\gamma'} \Rightarrow B_{\gamma'} \vdash_\varphi^f \text{abs}_{\beta\eta} g : B} \text{ (sub)}$$

Since $B\varphi = B$ holds, we are done.

Next, consider the third rule $f(\mathbf{app} \ x \ y) \rightarrow \mathbf{app}(f \ x)(f \ y)$. We define $\Gamma := \{x : B_{\gamma''}, y : B_{\gamma''}\}$, $B^x := B^y := B$, $\alpha^x := \alpha^y := \gamma''$ and $\varphi := \{(\alpha^f, \mathbf{s} \ \gamma'')\}$.

Monotony. Obvious.

Accessibility. We have $(x, B, B) \triangleleft_a (\mathbf{app} \ x \ y, B, B)$ and $(y, B, B) \triangleleft_a (\mathbf{app} \ x \ y, B, B)$.

Minimality. Let θ be a substitution with $\mathbf{app}(x\theta)(y\theta) \in B$. We define ν as $\nu := \{(\gamma'', \mathbf{b})\}$ with $\mathbf{b} := \max\{o_{S^B}(x\theta), o_{S^B}(y\theta)\}$. Then, we have $o_{S^B}(x\theta) \leq \alpha^x \nu$ and $o_{S^B}(y\theta) \leq \alpha^y \nu$. Moreover, by Lemma 13, we have $\alpha^f \varphi \nu = (\mathbf{s} \ \gamma'') \nu = \mathbf{b} + 1 = \Sigma^{\mathbf{app}}(o_{S^B}(x\theta), o_{S^B}(y\theta)) = o_{S^B}(\mathbf{app}(x\theta)(y\theta))$.

Subject Reduction and Decreasingness. Define $\psi := \{(\alpha^f, \gamma'')\}$. We have the following:

$$\frac{\frac{\Gamma \vdash_{\varphi}^f x : B_{\gamma''}}{\Gamma \vdash_{\varphi}^f f \ x : B_{\gamma''}} \text{ (app-decr)} \quad \frac{\Gamma \vdash_{\varphi}^f y : B_{\gamma''}}{\Gamma \vdash_{\varphi}^f f \ y : B_{\gamma''}} \text{ (app-decr)}}{\Gamma \vdash_{\varphi}^f \mathbf{app}(f \ x)(f \ y) : B_{\mathbf{s} \ \gamma''}} \text{ (app-decr)}$$

where $B_{\gamma''} = B_{\alpha^f \psi}$ and $(f, \psi) <_A (f, \varphi)$ holds.

5 Concluding Remarks and Future Work

We, in this paper, have extended the termination criterion in [9] so that in some case the termination of the rewrite relation induced by rewrite rules on non-positive types can be shown. For this purpose, the inflationary fixed-point construction has been used: the inflationary fixed-point construction is crucial to the definition of stratifications by size functions for non-positive types. In addition, we have also improved the criterion in [9] with regard to non-strictly positive inductive types. We have noted that a condition on a typing rule for the computability closure can be dropped, and then we have shown the termination of Hofmann's extract function for the breadth-first traversal of trees. This example is a typical case of rewrite systems on non-strictly positive inductive types.

However, a thorough study of rewrite rules on non-positive types is far from being achieved, since it is dependent type systems that are able to include more impressive examples of non-positive types. A larger goal is thus to extend our termination criterion to dependent type systems. Setzer's Mahlo universe ([23]), which is a universe type with a strong reflection property in Martin-Löf type theory, is an example of non-positive types in dependent type systems. Exploring rewrite rules for Setzer's Mahlo universe would be a crucial step for further research on combined systems of typed λ -calculus and rewrite rules. For this purpose, we will examine whether our criterion can be extended to $\lambda\Pi/\mathcal{R}$ -calculus, and whether Mahlo universe can be formulated in $\lambda\Pi/\mathcal{R}$ -calculus. This calculus is a combined system of the dependent type system $\lambda\Pi$ -calculus and rewrite rules. Some termination criteria for $\lambda\Pi/\mathcal{R}$ -calculus were already formulated by [10, 16], but, to the best of our knowledge, rewrite rules on non-positive types in this calculus remain unexplored.

Recently, dependently typed programming languages such as Coq and Agda were combined with rewrite rules ([13, 14]). Providing these combined languages with termination criteria would be another crucial step for further research on rewrite rules in dependent type systems, since rewrite rules and several features from Coq or Agda coexist there. Termination criteria on strictly or non-strictly positive inductive types in these languages are not sufficiently examined yet, so we are planning to begin by exploring positive inductive types. In particular, it should be investigated whether the size-based termination method is applicable to formulate termination criteria on positive inductive types in these combined programming languages.

Yet another larger goal is to integrate our work into an automated termination prover for higher-order rewriting such as Blanqui's HOT. Since HOT is based on sized types and computability closure, HOT is most relevant to our work among automated termination provers.

References

- 1 Andreas Abel. Type-based termination, inflationary fixed-points, and mixed inductive-coinductive types. In *Proceedings 8th Workshop on Fixed Points in Computer Science, FICS 2012, Tallinn, Estonia, 24th March 2012*, pages 1–11, 2012. doi:10.4204/EPTCS.77.1.
- 2 Andreas Abel and Brigitte Pientka. Well-founded recursion with copatterns and sized types. *J. Funct. Program.*, 26:e2, 2016. doi:10.1017/S0956796816000022.
- 3 Franz Baader and Tobias Nipkow. *Term Rewriting and All That*. Cambridge University Press, 1998. doi:10.1017/CB09781139172752.
- 4 Franco Barbanera, Maribel Fernández, and Herman Geuvers. Modularity of strong normalization in the algebraic-lambda-cube. *J. Funct. Program.*, 7(6):613–660, 1997. doi:10.1017/s095679689700289x.
- 5 Gilles Barthe and Herman Geuvers. Modular properties of algebraic type systems. In *Higher-Order Algebra, Logic, and Term Rewriting, Second International Workshop, HOA '95, Paderborn, Germany, September 21-22, 1995, Selected Papers*, pages 37–56, 1995. doi:10.1007/3-540-61254-8_18.
- 6 Gilles Barthe and Femke van Raamsdonk. Termination of algebraic type systems: The syntactic approach. In *Algebraic and Logic Programming, 6th International Joint Conference, ALP '97 - HOA '97, Southampton, UK, September 3-5, 1997, Proceedings*, pages 174–193, 1997. doi:10.1007/BFb0027010.
- 7 Frédéric Blanqui. Definitions by rewriting in the calculus of constructions. *Mathematical Structures in Computer Science*, 15(1):37–92, 2005. doi:10.1017/S0960129504004426.
- 8 Frédéric Blanqui. Inductive types in the calculus of algebraic constructions. *Fundamenta Informaticae*, 65(1-2):61–86, 2005.
- 9 Frédéric Blanqui. Size-based termination of higher-order rewriting. *J. Funct. Program.*, 28:e11, 2018. doi:10.1017/S0956796818000072.
- 10 Frédéric Blanqui, Guillaume Genestier, and Olivier Hermant. Dependency pairs termination in dependent type theory modulo rewriting. In *4th International Conference on Formal Structures for Computation and Deduction, FSCD 2019, June 24-30, 2019, Dortmund, Germany*, pages 9:1–9:21, 2019. doi:10.4230/LIPIcs.FSCD.2019.9.
- 11 Frédéric Blanqui, Jean-Pierre Jouannaud, and Mitsuhiro Okada. Inductive-data-type systems. *Theoretical Computer Science*, 272(1):41–68, 2002. doi:10.1016/S0304-3975(00)00347-9.
- 12 Val Breazu-Tannen and Jean Gallier. Polymorphic rewriting conserves algebraic strong normalization and confluence. In Giorgio Ausiello, Mariangiola Dezani-Ciancaglini, and Simonetta Ronchi Della Rocca, editors, *ICALP 1989: Automata, Languages and Programming*, pages 137–150, Berlin, Heidelberg, 1989. Springer Berlin Heidelberg. doi:10.1007/BFb0035757.
- 13 Jesper Cockx. Type theory unchained: Extending agda with user-defined rewrite rules. In *25th International Conference on Types for Proofs and Programs, TYPES 2019, June 11-14, 2019, Oslo, Norway*, pages 2:1–2:27, 2019. doi:10.4230/LIPIcs.TYPES.2019.2.
- 14 Jesper Cockx, Nicolas Tabareau, and Théo Winterhalter. The taming of the rew: a type theory with computational assumptions. *Proc. ACM Program. Lang.*, 5(POPL):1–29, 2021. doi:10.1145/3434341.
- 15 Carsten Fuhs and Cynthia Kop. Polynomial interpretations for higher-order rewriting. In *23rd International Conference on Rewriting Techniques and Applications (RTA '12), RTA 2012, May 28 - June 2, 2012, Nagoya, Japan*, pages 176–192, 2012. doi:10.4230/LIPIcs.RTA.2012.176.

- 16 Guillaume Genestier. *Dependently-Typed Termination and Embedding of Extensional Universe-Polymorphic Type Theory using Rewriting*. PhD thesis, University of Paris-Saclay, France, 2020. URL: <https://tel.archives-ouvertes.fr/tel-03167579>.
- 17 Jean-Pierre Jouannaud and Mitsuhiro Okada. Abstract data type systems. *Theoretical Computer Science*, 173(2):349–391, 1997. doi:10.1016/S0304-3975(96)00161-2.
- 18 Cynthia Kop and Femke van Raamsdonk. Dynamic dependency pairs for algebraic functional systems. *Log. Methods Comput. Sci.*, 8(2), 2012. doi:10.2168/LMCS-8(2:10)2012.
- 19 Ralph Matthes. *Lambda Calculus: A Case for Inductive Definitions*. Unpublished lecture notes, 2000.
- 20 Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1):159–172, 1991. doi:10.1016/0168-0072(91)90069-X.
- 21 Mitsuhiro Okada. Strong normalizability for the combined system of the typed lambda calculus and an arbitrary convergent term rewrite system. In *Proceedings of the ACM-SIGSAM 1989 International Symposium on Symbolic and Algebraic Computation, ISSAC '89, Portland, Oregon, USA, July 17-19, 1989*, pages 357–363, 1989. doi:10.1145/74540.74582.
- 22 Erik Palmgren. On universes in type theory. In Giovanni Sambin and Jan M. Smith, editors, *Twenty Five Years of Constructive Type Theory*, Oxford Logic Guides, pages 191–204. Oxford University Press, 1998.
- 23 Anton Setzer. Extending Martin-Löf type theory by one Mahlo-universe. *Arch. Math. Log.*, 39(3):155–181, 2000. doi:10.1007/s001530050140.
- 24 Christoph Sprenger and Mads Dam. On the structure of inductive reasoning: Circular and tree-shaped proofs in the μ -calculus. In *Foundations of Software Science and Computational Structures, 6th International Conference, FOSSACS 2003 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2003, Warsaw, Poland, April 7-11, 2003, Proceedings*, pages 425–440, 2003. doi:10.1007/3-540-36576-1_27.
- 25 Daria Walukiewicz-Chrzęszcz. Termination of rewriting in the calculus of constructions. *J. Funct. Program.*, 13(2):339–414, 2003. doi:10.1017/S0956796802004641.

