23rd International Conference on Types for Proofs and Programs

TYPES 2017, May 24 – June 1, 2017, Budapest, Hungary

Edited by

Andreas Abel
Fredrik Nordvall Forsberg
Ambrus Kaposi

LIPICS

Editors

Andreas Abel
Dept. of Computer Science and Engineering
Gothenburg University
`andreas.abel@gu.se`

Ambrus Kaposi
Faculty of Informatics
Eötvös Loránd University
`akaposi@inf.elte.hu`

Fredrik Nordvall Forsberg
Dept. of Computer and Information Sciences
University of Strathclyde
`fredrik.nordvall-forsberg@strath.ac.uk`

ACM Classification 2012
Theory of computation → Type theory, Theory of computation → Proof theory, Theory of computation → Program verification

ISBN 978-3-95977-071-2

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-95977-071-2.

Publication date
December, 2018

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 http://dnb.d-nb.de.

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2017.0

LIPIcs – Leibniz International Proceedings in Informatics

ISSN 1868-8969

http://www.dagstuhl.de/lipics

Contents

Preface

This volume is the post-proceedings of the 23rd International Conference on Types for Proofs and Programs, TYPES 2017, which was held at Eötvös Loránd University in Budapest, Hungary, between the 24th of May and 1st of June in 2017.

The TYPES meetings are a forum to present new and on-going work in all aspects of type theory and its applications, especially in formalized 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. Prior to the 2017 meeting in Budapest, TYPES meetings took place in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Torino (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal near Nijmegen (2002), Torino (2003), Jouy-en-Josas near Paris (2004), Nottingham (2006), Cividale del Friuli (2007), Torino (2008), Aussois (2009), Warsaw (2010), Bergen (2011), Toulouse (2013), Paris (2014), Tallinn (2015), and Novi Sad (2016), with post-proceedings published in various outlets, with the last five in LIPIcs.

The TYPES areas of interest include, but are not limited to: foundations of type theory and constructive mathematics; 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.

The TYPES conferences are traditionally of an open and informal character. Selection of talks for presentation at the conference is based on short abstracts — reporting on work in progress or work presented or published elsewhere is welcome. A formal, fully reviewed post-proceedings volume of unpublished work is prepared after the conference. The programme of TYPES 2017 included invited talks by Edwin Brady (University of St Andrews) on *An architecture for dependently typed applications in Idris* and Jakob Rehof (Technical University of Dortmund) on *Bounding principles for decision problems with intersection types*. The contributed part of the programme consisted of 50 talks, and the conference was attended by more than 82 researchers.

TYPES 2017 was sponsored by the COST Action EUTypes, supported by COST (European Cooperation in Science and Technology) and by Eötvös Loránd University.

The call for contributions to the post-proceedings of TYPES 2017 was open and not restricted to the authors and presentations of the conference. Out of 13 submitted papers, 7 were selected after a reviewing process where each paper was reviewed by between two and four referees; the final decisions were made by the editors. The papers span a wide range of interesting topics: linear logic, decidability of Hilbert-style axiomatisations, natural deduction, formalized proof systems, program verification for an ML-like language, a new axiomatisation of homotopy type theory, and category theory in type theory. We thank both the authors and the reviewers for their hard work.

Andreas Abel, Fredrik Nordvall Forsberg, and Ambrus Kaposi
Nottingham, October 2018

List of Authors

Guillaume Allais
Radboud University Nijmegen
The Netherlands
gallais@cs.ru.nl

Andrej Dudenhefner
Technical University of Dortmund
Germany
andrej.dudenhefner@cs.tu-dortmund.de

Herman Geuvers
Radboud University Nijmegen & Technical University Eindhoven
The Netherlands
herman@cs.ru.nl

Tonny Hurkens
The Netherlands
hurkens@science.ru.nl

Rodolphe Lepigre
Université Savoie Mont Blanc & Université Paris-Saclay
France
rodolphe.lepigre@inria.fr

Julius Michaelis
Technische Universität München
Germany
j.michaelis@tum.de

Tobias Nipkow
Technische Universität München
Germany
nipkow@in.tum.de

Ian Orton
University of Cambridge
United Kingdom
rio22@cam.ac.uk

Erik Palmgren
Stockholm University
Sweden
palmgren@math.su.se

Andrew M. Pitts
University of Cambridge
United Kingdom
andrew.pitts@cl.cam.ac.uk

Jakob Rehof
Technical University of Dortmund
Germany
jakob.rehof@cs.tu-dortmund.de

Typing with Leftovers – A mechanization of Intuitionistic Multiplicative-Additive Linear Logic

Guillaume Allais
iCIS, Radboud University Nijmegen, The Netherlands
gallais@cs.ru.nl

Abstract

We start from an untyped, well-scoped λ-calculus and introduce a bidirectional typing relation corresponding to a Multiplicative-Additive Intuitionistic Linear Logic. We depart from typical presentations to adopt one that is well-suited to the intensional setting of Martin-Löf Type Theory. This relation is based on the idea that a linear term consumes some of the resources available in its context whilst leaving behind leftovers which can then be fed to another program.

Concretely, this means that typing derivations have both an input and an output context. This leads to a notion of weakening (the extra resources added to the input context come out unchanged in the output one), a rather direct proof of stability under substitution, an analogue of the frame rule of separation logic showing that the state of unused resources can be safely ignored, and a proof that typechecking is decidable. Finally, we demonstrate that this alternative formalization is sound and complete with respect to a more traditional representation of Intuitionistic Linear Logic.

The work has been fully formalised in Agda, commented source files are provided as additional material available at `https://github.com/gallais/typing-with-leftovers`.

2012 ACM Subject Classification Theory of computation → Type theory, Theory of computation → Linear logic

Keywords and phrases Type System, Bidirectional, Linear Logic, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.1

Supplement Material `https://github.com/gallais/typing-with-leftovers`

Funding The research leading to these results has received funding from the European Research Council under the European Union's Seventh Framework Programme (FP7/2007-2013) / ERC grant agreement nr. 320571

1 Introduction

The strongly-typed functional programming community has benefited from a wealth of optimisations made possible precisely because the library author as well as the compiler are aware of the type of the program they are working on. These optimisations have ranged from Danvy's type-directed partial evaluation [18] residualising specialised programs to erasure mechanisms –be they user-guided like Coq's extraction [27] which systematically removes all the purely logical proofs put in Prop by the developer or automated like Brady and Tejiščák's erased values [12, 13]– and including the library defining the State-Thread [25] monad which relies on higher-rank polymorphism and parametricity to ensure the safety of using an actual mutable object in a lazy, purely functional setting.

However, in the context of the rising development of dependently-typed programming languages [11, 31] which, unlike ghc's Haskell [36], incorporate a hierarchy of universes in order to ensure that the underlying logic is consistent, some of these techniques are not applicable anymore. Indeed, the use of large quantification in the definition of the ST-monad crucially relies on impredicativity. As a consequence, the specification of programs allowed to update a mutable object in a safe way has to change. Idris has been extended with experimental support for uniqueness types inspired by Clean's [1] and Rust's ownership types [20], all of which stem from linear logic [22].

In order to be able to use type theory to formally study the meta-theory of the programming languages whose type system includes notions of linearity, we need to have a good representation of such constraints.

Section 2 introduces the well-scoped untyped λ-calculus we are going to use as our language of raw terms. Section 3 defines the linear typing rules for this language as relations which record the resources consumed by a program. The next sections are dedicated to proving properties of this type system: Section 4 proves that the status of unused variables rightfully does not matter, Section 5 (and respectively Section 6) demonstrates that these typing relations are stable under weakening (respectively substitution), Section 7 demonstrates that these relations are functional, and Section 8 that they are decidable i.e. provides us with a typechecking algorithm. Finally Section 9 goes back to a more traditional presentation of Intuitionistic Multiplicative-Additive Linear Logic and demonstrates it is equivalent to our type system.

Notations

This whole development has been fully formalised in Agda. Rather than using Agda's syntax, the results are reformulated in terms of definitions, lemmas, theorems, and examples. However it is important to keep in mind the distinction between various kinds of objects. `Teletype` is used to denote data constructors, SMALL CAPITALS are characteristic of defined types. A type family's index is written as a subscript e.g. VAR_n.

We use two kinds of inference rules to describe inductive families: double rules are used to define types whilst simple ones correspond to constructors. In each case the premises correspond to arguments (usually called parameters and indices for types) and the conclusion shows the name of the constructor. A typical example is the inductively defined set of unary natural numbers. The inductive type is called NAT and it has two constructors: 0 takes no argument whilst 1+· takes a NAT n and represents its successor.

$$\frac{}{\text{NAT} : \mathit{Set}} \qquad \frac{}{\texttt{0} : \text{NAT}} \qquad \frac{n : \text{NAT}}{\texttt{1+}n : \text{NAT}}$$

2 The Calculus of Raw Terms

The calculus we study in this paper is meant to be a core language, even though it will be rather easy to write programs in it. As a consequence all the design choices have been guided by the goal of facilitating its mechanical treatment in a dependently-typed language. That is why we use de Bruijn indices to represent variable bindings. We demonstrate in the code accompanying the paper how to combine a parser and a scope checker to turn a surface level version of the language using strings as variable names into this representation.

Following Bird and Patterson [9] and Altenkirch and Reus [5], we define the raw terms of our language not as an inductive type but rather as an inductive *family* [21]. This technique, sometimes dubbed "type-level de Bruijn indices", makes it possible to keep track, in the index of the family, of the free variables currently in scope. As is nowadays folklore, instead

$$
\begin{array}{lcl}
\langle \mathrm{INFER}_n \rangle & ::= & \texttt{var}\ \langle \mathrm{VAR}_n \rangle \\
& | & \texttt{app}\ \langle \mathrm{INFER}_n \rangle\ \langle \mathrm{CHECK}_n \rangle \\
& | & \texttt{case}\ \langle \mathrm{INFER}_n \rangle\ \texttt{return}\ \langle \mathrm{TYPE} \rangle\ \texttt{of}\ \langle \mathrm{CHECK}_{1+n} \rangle\ \texttt{\%\%}\ \langle \mathrm{CHECK}_{1+n} \rangle \\
& | & \texttt{prj}_1\ \langle \mathrm{INFER}_n \rangle\ \ |\ \ \texttt{prj}_2\ \langle \mathrm{INFER}_n \rangle \\
& | & \texttt{exfalso}\ \langle \mathrm{TYPE} \rangle\ \langle \mathrm{INFER}_n \rangle \\
& | & \texttt{cut}\ \langle \mathrm{CHECK}_n \rangle\ \langle \mathrm{TYPE} \rangle \\
\\
\langle \mathrm{CHECK}_n \rangle & ::= & \texttt{lam}\ \langle \mathrm{CHECK}_{1+n} \rangle \\
& | & \texttt{let}\ \langle \mathrm{PATTERN}_m \rangle \texttt{:=} \langle \mathrm{INFER}_n \rangle \texttt{in} \langle \mathrm{CHECK}_{m+n} \rangle \\
& | & \texttt{unit} \\
& | & \texttt{inl}\ \langle \mathrm{CHECK}_n \rangle\ \ |\ \ \texttt{inr}\ \langle \mathrm{CHECK}_n \rangle \\
& | & \texttt{prd}\ \langle \mathrm{CHECK}_n \rangle\ \langle \mathrm{CHECK}_n \rangle \\
& | & \texttt{neu}\ \langle \mathrm{INFER}_n \rangle
\end{array}
$$

Figure 1 Grammar of the Language of Raw Terms.

of using a set-indexed presentation where a closed terms is indexed by the empty set $\bot$ and fresh variables are introduced by wrapping the index in a `Maybe` type constructor[1], we index our terms by a natural number instead. The VAR type family[2] defined below represents the de Bruijn indices [19] corresponding to the n free variables present in a scope n.

$$\frac{n : \mathrm{NAT}}{\mathrm{VAR}_n : \mathit{Set}} \qquad \frac{}{\texttt{z} : \mathrm{VAR}_{1+n}} \qquad \frac{k : \mathrm{VAR}_n}{\texttt{s}\ k : \mathrm{VAR}_{1+n}}$$

We present the calculus in a bidirectional fashion [32]. This definition style scales well to more complex type theories where full type-inference is not tractable anymore whilst keeping the type annotations the programmer needs to add to a minimum. The term constructors of the calculus are split in two different syntactic categories corresponding to constructors of canonical values on one hand and eliminators on the other. These categories characterise the flow of information during typechecking: given a context assigning a type to each free variable, canonical values (which we call CHECK) can be *check*ed against a type whilst we may *infer* the type of computations (which we call INFER). Each type is indexed by a scope:

$$\frac{n : \mathrm{NAT}}{\mathrm{INFER}_n : \mathit{Set}} \qquad \frac{n : \mathrm{NAT}}{\mathrm{CHECK}_n : \mathit{Set}}$$

On top of the constructors one would expect for a usual definition of the untyped λ-calculus (`var ·`, `app · ·`, and `lam ·`) we have constructors and eliminators for sums (`inl ·`, `inr ·`, `case · return · of · %% ·`), products (`prd · ,` `let · := · in ·`, `prj₁ ·`, `prj₂ ·`), unit (`unit`, `let · := · in ·`) and void (`exfalso · ·`). Two additional rules (`neu ·` and `cut · ·` respectively) allow the embedding of INFER into CHECK and vice-versa. They make it possible to form redexes by embedding canonical values into computations and then applying eliminators to them. In terms of typechecking, they correspond to a change of direction between inferring and checking.

[1] The value `nothing` represents the fresh variable whilst the constructor `just` lifts the other ones in the new scope.
[2] It is also known as `Fin` (for "finite set") in the dependently typed programming community.

$$
\begin{array}{lcl}
\langle\text{TYPE}\rangle & ::= & \kappa\ \langle\mathbb{N}\rangle\ \mid\ \mathbb{0}\ \mid\ \mathbb{1} \\
 & \mid & \langle\text{TYPE}\rangle\ \multimap\ \langle\text{TYPE}\rangle\ \mid\ \langle\text{TYPE}\rangle\ \otimes\ \langle\text{TYPE}\rangle \\
 & \mid & \langle\text{TYPE}\rangle\ \oplus\ \langle\text{TYPE}\rangle\ \mid\ \langle\text{TYPE}\rangle\ \&\ \langle\text{TYPE}\rangle
\end{array}
$$

Figure 2 Grammar of TYPE.

The constructors `cut`, `case`, and `exfalso` take an extra TYPE argument in order to guarantee the success and uniqueness of type-inference for INFER terms.

A notable specificity of this language is the ability to use nested patterns in a let binder rather than having to resort to cascading lets. This is achieved thanks to a rather simple piece of kit: the PATTERN type family. A value of type PATTERN_n represents an irrefutable pattern binding n variables. Because variables are represented as de Bruijn indices, the base pattern does not need to be associated with a name, it simply is a constructor `v` binding exactly one variable. The brackets pattern $\langle\rangle$ matches unit values and binds nothing. The comma pattern constructor $(\cdot\ \texttt{,}\ \cdot)$ takes two nested patterns respectively binding m and n variables and uses them to deeply match a pair thus binding $(m + n)$ variables.

$$
\frac{n : \text{NAT}}{\text{PATTERN}_n : \mathit{Set}}
\qquad
\frac{}{\texttt{v} : \text{PATTERN}_1}
\qquad
\frac{}{\langle\rangle : \text{PATTERN}_0}
\qquad
\frac{p : \text{PATTERN}_m \qquad q : \text{PATTERN}_n}{(p\ \texttt{,}\ q) : \text{PATTERN}_{m+n}}
$$

The grammar of raw terms only guarantees that all expressions are well-scoped by construction. It does not impose any other constraint, which means that a user may write valid programs but also invalid ones as the following examples demonstrate:

▶ **Example 1.** `swap` is a closed, well-typed linear term taking a pair as an input and swapping its components. It corresponds to the mathematical function $(x, y) \mapsto (y, x)$.

```
swap = lam (let (v , v) := var z
            in prd (neu (var (s z))) (neu (var z)))
```

▶ **Example 2.** `illTyped` is a closed linear term. However it is manifestly ill-typed: the let-binding it uses tries to break down a function as if it were a pair.

```
illTyped = let (v , v) := cut (lam (neu (var z))) (a ⊸ a)
           in prd (neu (var z)) (neu (var (s z)))
```

▶ **Example 3.** Finally, `diagonal` is a term typable in the simply-typed lambda calculus but it is not linear: it duplicates its input just like $x \mapsto (x, x)$ does.

```
diagonal = lam (prd (neu (var z)) (neu (var z)))
```

3 Linear Typing Rules

These examples demonstrate that we need to define a typing relation describing the rules terms need to abide by in order to qualify as well-typed linear programs. We start by defining the types our programs may have using the grammar in Figure 2. Apart from the usual linear type formers, we have a constructor κ which makes it possible to have countably many different base types.

A linear type system is characterised by the fact that all the resources available in the context have to be used exactly once by the term being checked. In traditional presentations

$$\frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau}{\Gamma, \Delta \vdash \sigma \otimes \tau}\otimes_i \qquad\qquad \frac{\Gamma \vdash \sigma \qquad \Delta \vdash \tau \qquad \Gamma, \Delta \simeq \Theta}{\Theta \vdash \sigma \otimes \tau}\otimes_i$$

Figure 3 Introduction rules for tensor (left: usual presentation, right: with reordering on the fly).

of linear logic this is achieved by representing the context as a multiset and, in each rule, cutting it up and distributing its parts among the premises. This is epitomised by the introduction rule for tensor.

However, multisets are an intrinsically *extensional* notion and therefore quite arduous to work with in an *intensional* type theory. Various strategies can be applied to tackle this issue; most of them rely on using linked lists to represent contexts together with ways to reorganise the context.

In Figure 3 we show two of the most common representations of the tensor rule. The first one splits the context into Γ and Δ and dispatches them into the subproofs; it relies on the existence of structural rules which the user will be able to use to reorganise the context appropriately. The second one is a combined rule letting the user re-arrange the context on the fly by using the notion of “bag-equivalence” for lists denoted $\cdot \simeq \cdot$.

In both of these cases, the user has to explicitly rearrange the context either by using structural rules or proving that two distinct contexts are bag equivalent. Although one can find coping mechanisms to handle such clunky systems (for instance using a solver for bag-equivalence [17] based on the proof-by-reflection [10] approach to automation), we would rather not.

All of these strategies are artefacts of the unfortunate mismatch between the ideal mathematical objects one wishes to model and their internal representation in the proof assistant. Short of having proper quotient types, this will continue to be an issue when dealing with multisets. The solution described in the rest of this paper is syntax-directed; it does not try to replicate a set-theoretic approach in intuitionistic type theory but rather strives to find the type theoretical structures which can make the problem more tractable. Indeed, given the right abstractions most proofs become direct structural inductions.

3.1 Usage Annotations

McBride’s recent work [29] on combining linear and dependent types highlights the distinction one can make between referring to a resource and actually consuming it. In the same spirit, rather than dispatching the available resources in the appropriate subderivations, we consider that a term is checked in a *given* context on top of which usage annotations are super-imposed. These usage annotations indicate whether resources have been consumed already or are still available. Type-inference (resp. Type-checking) is then inferring (resp. checking) a term’s type but *also* annotating the resources consumed by the term in question and returning the *leftovers* which gave their name to this paper.

▶ **Definition 4.** A **CONTEXT** is a list of TYPEs indexed by its length. It can be formally described by the following inference rules:

$$\frac{n : \text{NAT}}{\text{CONTEXT}_n : \mathit{Set}} \qquad \frac{}{[] : \text{CONTEXT}_0} \qquad \frac{\gamma : \text{CONTEXT}_n \qquad \sigma : \text{TYPE}}{\gamma \cdot \sigma : \text{CONTEXT}_{1+n}}$$

▶ **Definition 5.** A **USAGE** is a predicate on a type σ describing whether the resource associated to it is available or not. We name the constructors describing these two states `f` (for **fresh**) and `s` (for **stale**) respectively. These are naturally lifted to contexts in a pointwise manner and we reuse the USAGE name and the `f` and `s` names for the functions taking a context and returning either a fully fresh or fully stale USAGE for it.

$$\frac{\sigma : \mathrm{TYPE}}{\mathrm{USAGE}_\sigma : \mathit{Set}} \qquad \frac{}{\mathtt{f}_\sigma : \mathrm{USAGE}_\sigma} \qquad \frac{}{\mathtt{s}_\sigma : \mathrm{USAGE}_\sigma}$$

$$\frac{\gamma : \mathrm{CONTEXT}_n}{\mathrm{USAGE}_\gamma : \mathit{Set}} \qquad \frac{}{[] : \mathrm{USAGE}_{[]}} \qquad \frac{\Gamma : \mathrm{USAGE}_\gamma \qquad S : \mathrm{USAGE}_\sigma}{\Gamma \cdot S : \mathrm{USAGE}_{\gamma \cdot \sigma}}$$

3.2 Typing as Consumption Annotation

A Typing relation seen as a consumption annotation process describes what it means to analyze a term in a scope of size n: given a context of types for these n variables, and a usage annotation for that context, it ascribes a type to the term whilst crafting another usage annotation containing all the leftover resources. Formally:

▶ **Definition 6.** A **Typing Relation** for T a NAT-indexed inductive family is an indexed relation $\mathcal{T}_n$ such that:

$$\frac{n : \mathrm{NAT} \qquad \gamma : \mathrm{CONTEXT}_n \qquad \Gamma, \Delta : \mathrm{USAGE}_\gamma \qquad t : T_n \qquad \sigma : \mathrm{TYPE}}{\mathcal{T}_n(\Gamma, t, \sigma, \Delta) : \mathit{Set}}$$

This definition clarifies the notion but also leads to more generic statements later on: weakening, substitution, framing can all be expressed as properties a Typing Relation might have. We can already list the typing relations introduced later on in this article which fit this pattern. We have split their arguments into three columns depending on whether they should be understood as either inputs (the known things), scrutinees (the things being validated), or outputs (the things that we learn) and hint at what the flow of information in the typechecker will be.

▶ Remark. The use of $\boxtimes$ is meant to suggest that the input Γ gets distributed between the type σ of the term and the leftovers Δ obtained as an output. Informally $\Gamma \simeq \sigma \otimes \Delta$, hence the use of a tensor-like symbol.

3.2.1 Typing de Bruijn indices

The simplest instance of a Typing Relation is the one for de Bruijn indices: given an index k and a usage annotation, it successfully associates a type σ to that index if and only if the kth resource in context is of type σ and fresh (i.e. its USAGE_σ is $\mathtt{f}_\sigma$). In the resulting leftovers, this resource will have turned stale ($\mathtt{s}_\sigma$) because it has now been used:

▶ **Definition 7.** The typing relation for VAR is presented in a sequent-style: $\Gamma \vdash_v k \in \sigma \boxtimes \Delta$ means that starting from the usage annotation Γ, the de Bruijn index k is ascribed type σ with leftovers Δ. It is defined inductively by two constructors (cf. Figure 5).

▶ Remark. The careful reader will have noticed that there is precisely one typing rule for each VAR constructor. It is not a coincidence. And if these typing rules are not named it's because in Agda, they can be given the same name as their VAR counterpart and the

$$\frac{\gamma : \text{CONTEXT}_n \quad \Gamma : \text{USAGE}_\gamma \quad k : \text{VAR}_n \quad \sigma : \text{TYPE} \quad \Delta : \text{USAGE}_\gamma}{\Gamma \vdash_v k \in \sigma \boxtimes \Delta : \mathit{Set}}\,\text{VAR}$$

$$\frac{\gamma : \text{CONTEXT}_n \quad \Gamma : \text{USAGE}_\gamma \quad t : \text{INFER}_n \quad \sigma : \text{TYPE} \quad \Delta : \text{USAGE}_\gamma}{\Gamma \vdash t \in \sigma \boxtimes \Delta : \mathit{Set}}\,\text{INFER}$$

$$\frac{\gamma : \text{CONTEXT}_n \quad \Gamma : \text{USAGE}_\gamma \quad \sigma : \text{TYPE} \quad t : \text{CHECK}_n \quad \Delta : \text{USAGE}_\gamma}{\Gamma \vdash \sigma \ni t \boxtimes \Delta : \mathit{Set}}\,\text{CHECK}$$

$$\frac{\sigma : \text{TYPE} \quad p : \text{PATTERN}_n \quad \gamma : \text{CONTEXT}_n}{\sigma \ni p \rightsquigarrow \gamma : \mathit{Set}}\,\text{PATTERN}$$

Figure 4 Typing relations for VAR, INFER, CHECK and PATTERN.

$$\frac{}{\Gamma \cdot \mathtt{f}_\sigma \vdash_v \mathtt{z} \in \sigma \boxtimes \Gamma \cdot \mathtt{s}_\sigma} \qquad \frac{\Gamma \vdash_v k \in \sigma \boxtimes \Delta}{\Gamma \cdot A \vdash_v \mathtt{s}\, k \in \sigma \boxtimes \Delta \cdot A}$$

Figure 5 Typing rules for VAR.

typechecker will perform type-directed disambiguation. The same will be true for INFER, CHECK and PATTERN which means that writing down a typable program could be seen as either writing a raw term or the typing derivation associated to it depending on the author's intent.

▶ **Example 8.** The de Bruijn index 1 has type τ in the context $(\gamma \cdot \tau \cdot \sigma)$ with usage annotation $(\Gamma \cdot \mathtt{f}_\tau \cdot \mathtt{f}_\sigma)$, no matter what Γ actually is:

$$\frac{\dfrac{}{\Gamma \cdot \mathtt{f}_\tau \vdash \mathtt{z} \in \tau \boxtimes \Gamma \cdot \mathtt{s}_\tau}}{\Gamma \cdot \mathtt{f}_\tau \cdot \mathtt{f}_\sigma \vdash \mathtt{s}\,\mathtt{z} \in \tau \boxtimes \Gamma \cdot \mathtt{s}_\tau \cdot \mathtt{f}_\sigma}$$

Or, as it would be written in Agda, taking advantage of the fact that the language constructs and the typing rules about them have been given the same names:

```
one : Γ · f τ · f σ ⊢ s z ∈ τ ⊠ Γ · s τ · f σ
one = s z
```

3.2.2 Typing Terms

We now face compound untyped terms such as `app` f t whose subterms f and t have been defined in the *same* scope of size n. Therefore the typing relation for these terms needs to

$$\frac{\Gamma \vdash_v k \in \sigma \boxtimes \Delta}{\Gamma \vdash \texttt{var}\ k \in \sigma \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \sigma \multimap \tau \boxtimes \Delta \qquad \Delta \vdash \sigma \ni u \boxtimes \Theta}{\Gamma \vdash \texttt{app}\ t\ u \in \tau \boxtimes \Theta}$$

$$\frac{\Gamma \vdash t \in \sigma \oplus \tau \boxtimes \Delta \qquad \begin{array}{c} \Delta \cdot \texttt{f}_\sigma \vdash \nu \ni l \boxtimes \Theta \cdot \texttt{s}_\sigma \\ \Delta \cdot \texttt{f}_\tau \vdash \nu \ni r \boxtimes \Theta \cdot \texttt{s}_\tau \end{array}}{\Gamma \vdash \texttt{case}\ t\ \texttt{return}\ \nu\ \texttt{of}\ l \mathbin{\%} r \in \nu \boxtimes \Theta} \qquad \frac{\Gamma \vdash t \in \sigma \& \tau \boxtimes \Delta}{\Gamma \vdash \texttt{prj}_1\ t \in \sigma \boxtimes \Delta}$$

$$\frac{\Gamma \vdash t \in \sigma \& \tau \boxtimes \Delta}{\Gamma \vdash \texttt{prj}_2\ t \in \tau \boxtimes \Delta} \qquad \frac{\Gamma \vdash t \in \mathbb{0} \boxtimes \Delta}{\Gamma \vdash \texttt{exfalso}\ \sigma\ t \in \sigma \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \texttt{cut}\ t\ \sigma \in \sigma \boxtimes \Delta}$$

Figure 6 Typing rules for INFER.

use the *same* context of size n for both premises. Trying to cut up a CONTEXT$_n$ in two just like in Figure 3 would not only be cumbersome, it wouldn't be type correct. This is where usage annotations shine.

The key idea appearing in all the typing rules for compound expressions is to use the input USAGE to type one of the sub-expressions, collect the leftovers from that typing derivation and use them as the new input USAGE when typing the next sub-expression.

Another common pattern can be seen across all the rules involving binders, be they λ-abstractions, let-bindings or branches of a case. Typechecking the body of a binder involves extending the input USAGE with fresh variables and observing that they have become stale in the output one. This guarantees that these bound variables cannot escape their scope as well as that they have indeed been used. Although not the focus of this paper, it is worth noting that relaxing the staleness restriction would lead to an affine type system which would be interesting in its own right.

▶ **Definition 9.** The Typing Relation for INFER is typeset in a fashion similar to the one for VAR: in both cases the type is inferred. $\Gamma \vdash t \in \sigma \boxtimes \Delta$ means that given Γ a USAGE$_\gamma$, and t an INFER, the type σ is inferred together with leftovers Δ, another USAGE$_\gamma$. The rules are listed in Figure 6.

▶ **Definition 10.** For CHECK, the type σ comes first: $\Gamma \vdash \sigma \ni \mathrm{t} \boxtimes \Delta$ means that given Γ a USAGE$_\gamma$, a type σ, the CHECK t can be checked to have type σ with leftovers Δ. The rules can be found in Figure 7.

We can see that both variants of a product type –tensor ($\otimes$) and with ($\&$)– use the same surface language constructor but are disambiguated in a type-directed manner in the checking relation. The premises are naturally widely different: With lets its user pick which of the two available types they want and as a consequence both components have to be proven using the same resources. Tensor on the other hand forces the user to use both so the leftovers are threaded from one premise to the other.

▶ **Definition 11.** Finally, PATTERNs are checked against a type and a context of newly bound variables is generated. If the variable pattern always succeeds, the pair constructor pattern on the other hand only succeeds if the type it attempts to split is a tensor type. The context of newly-bound variables is then the collection of the contexts associated to the nested patterns. The rules are given in Figure 8.

$$\frac{\Gamma \cdot \mathtt{f}_\sigma \vdash \tau \ni b \boxtimes \Delta \cdot \mathtt{s}_\sigma}{\Gamma \vdash \sigma \multimap \tau \ni \mathtt{lam}\ b \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Delta \vdash \tau \ni b \boxtimes \Theta}{\Gamma \vdash \sigma \otimes \tau \ni \mathtt{prd}\ a\ b \boxtimes \Theta}$$

$$\frac{\Gamma \vdash \sigma \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \mathtt{inl}\ t \boxtimes \Delta} \qquad \frac{\Gamma \vdash \tau \ni t \boxtimes \Delta}{\Gamma \vdash \sigma \oplus \tau \ni \mathtt{inr}\ t \boxtimes \Delta} \qquad \frac{\Gamma \vdash \sigma \ni a \boxtimes \Delta \qquad \Gamma \vdash \tau \ni b \boxtimes \Delta}{\Gamma \vdash \sigma \& \tau \ni \mathtt{prd}\ a\ b \boxtimes \Delta}$$

$$\frac{}{\Gamma \vdash \mathbb{1} \ni \mathtt{unit} \boxtimes \Gamma} \qquad \frac{\begin{array}{c}\Gamma \vdash t \in \sigma \boxtimes \Delta \qquad \sigma \ni p \rightsquigarrow \delta \\ (\Delta \mathbin{+\!\!+} \mathtt{f}_\delta) \vdash \tau \ni u \boxtimes (\Theta \mathbin{+\!\!+} \mathtt{s}_\delta)\end{array}}{\Gamma \vdash \tau \ni \mathtt{let}\ p \mathbin{:=} t \mathtt{in}\ u \boxtimes \Theta} \qquad \frac{\Gamma \vdash t \in \sigma \boxtimes \Delta}{\Gamma \vdash \sigma \ni \mathtt{neu}\ t \boxtimes \Delta}$$

Figure 7 Typing rules for CHECK.

$$\frac{}{\sigma \ni \mathtt{v} \rightsquigarrow [] \cdot \sigma} \qquad \frac{}{\mathbb{1} \ni \langle\rangle \rightsquigarrow []} \qquad \frac{\sigma \ni p \rightsquigarrow \gamma \qquad \tau \ni q \rightsquigarrow \delta}{\sigma \otimes \tau \ni (p\ ,\ q) \rightsquigarrow (\delta \mathbin{+\!\!+} \gamma)}$$

Figure 8 Typing rules for PATTERN.

▶ **Example 12.** Given these rules, we see that the identity function can be checked at type $(\sigma \multimap \sigma)$ in an empty context:

$$\cfrac{\cfrac{\cfrac{\cfrac{}{[] \cdot \mathtt{f}_\sigma \vdash_v \mathtt{z} \in \sigma \boxtimes [] \cdot \mathtt{s}_\sigma}}{[] \cdot \mathtt{f}_\sigma \vdash \mathtt{var}\ \mathtt{z} \in \sigma \boxtimes [] \cdot \mathtt{s}_\sigma}}{[] \cdot \mathtt{f}_\sigma \vdash \sigma \ni \mathtt{neu}\ (\mathtt{var}\ \mathtt{z}) \boxtimes [] \cdot \mathtt{s}_\sigma}}{[] \vdash \sigma \multimap \sigma \ni \mathtt{lam}\ (\mathtt{neu}\ (\mathtt{var}\ \mathtt{z})) \boxtimes []}$$

Or, as it would be written in Agda where the typing rules were given the same name as their term constructor counterparts:

```
identity : [] ⊢ σ ⊸ σ ∋ lam (neu (var z)) ⊠ []
identity = lam (neu (var z))
```

▶ **Example 13.** It is also possible to revisit Example 1 to prove that `swap` can be checked against type $(\sigma \otimes \tau) \multimap (\tau \otimes \sigma)$ in an empty context. This gives the lengthy derivation included in the appendix or the following one in Agda which is quite a lot more readable:

```
swapTyped : [] ⊢ (σ ⊗ τ) ⊸ (τ ⊗ σ) ∋ swap ⊠ []
swapTyped = lam (let (v , v) := var z
                 in prd (neu (var (s z))) (neu (var z))
```

4 Framing

The most basic property one can prove about this typing system is the fact that the state of the resources which are not used by a lambda term is irrelevant. We call this property the Framing Property because of the obvious analogy with the frame rule in separation logic. This can be reformulated as the fact that as long as two pairs of an input and an output USAGE exhibit the same consumption pattern then if a derivation uses one of these, it can use the other one instead. Formally (postponing the definition of $\Gamma - \Delta \equiv \Theta - \Xi$):

▶ **Definition 14.** A Typing Relation $\mathcal{T}$ for a NAT-indexed family T has the **Framing Property** if for all k a NAT, γ a CONTEXT$_k$, Γ, Δ, Θ, Ξ four USAGE$_\gamma$, t an element of T_k and σ a Type, if $\Gamma - \Delta \equiv \Theta - \Xi$ and $\mathcal{T}_k(\Gamma,\ \mathrm{t},\ \sigma,\ \Delta)$ then $\mathcal{T}_k(\Theta,\ \mathrm{t},\ \sigma,\ \Xi)$ also holds.

▶ Remark. This is purely a property of the type system as witnessed by the fact that the term t is left unchanged wich won't be the case when defining stability under Weakening or Substitution for instance.

▶ **Definition 15. Consumption Equivalence** for a given CONTEXT γ characterises the pairs of an input and an output USAGE$_\gamma$ which have the same consumption pattern. The usages annotations for the empty context are trivially related. If the context is not empty, then there are two cases: if the resource is left untouched on one side, then so should it on the other side but the two annotations may be different (here denoted A and B respectively). On the other hand, if the resource has been consumed on one side then it has to be on the other side too.

$$\frac{\Gamma, \Delta, \Theta, \Xi : \mathrm{USAGE}_\gamma}{\Gamma - \Delta \equiv \Theta - \Xi : \mathit{Set}} \qquad \frac{}{[] - [] \equiv [] - []}$$

$$\frac{\Gamma - \Delta \equiv \Theta - \Xi}{(\Gamma \cdot A) - (\Delta \cdot A) \equiv (\Theta \cdot B) - (\Xi \cdot B)} \qquad \frac{\Gamma - \Delta \equiv \Theta - \Xi}{(\Gamma \cdot \mathtt{f}_\sigma) - (\Delta \cdot \mathtt{s}_\sigma) \equiv (\Theta \cdot \mathtt{f}_\sigma) - (\Xi \cdot \mathtt{s}_\sigma)}$$

▶ Remark. Two pairs of usages which are consumption equivalent are defined over the *same* context (and thus scope). Stability of a typing rule with respect to consumption equivalence will not be sufficient to introduce new variables. This will be dealth with by defining weakening in Section 5.

▶ **Definition 16.** The **Consumption Partial Order** $\Gamma \subseteq \Delta$ is defined as $\Gamma - \Delta \equiv \Gamma - \Delta$. It orders USAGE from least consumed to maximally consumed.

▶ **Lemma 17.** *The following properties on the Consumption relations hold:*
1. *The consumption equivalence is a partial equivalence [30].*
2. *The consumption partial order is a partial order.*
3. *If there is a USAGE Ξ "in between" two others Γ and Δ according to the consumption partial order (i.e. $\Gamma \subseteq \Xi$ and $\Xi \subseteq \Delta$), then any pair of USAGE Θ, Ω consumption equal to Γ and Δ (i.e. $\Gamma - \Delta \equiv \Theta - \Omega$) can be split in a manner compatible with Ξ. In other words: one can find Z such that $\Gamma - \Xi \equiv \Theta - Z$ and $\Xi - \Delta \equiv Z - \Omega$.*

▶ **Lemma 18** (Consumption). *The Typing Relations for VAR, INFER and CHECK all imply that if a typing derivation exists with input USAGE annotation Γ and output USAGE annotation Δ then $\Gamma \subseteq \Delta$.*

▶ **Theorem 19.** *The Typing Relation for* VAR *has the Framing Property. So do the ones for* INFER *and* CHECK*.*

Proof. The proofs are by structural induction on the typing derivations. They rely on the previous lemmas to, when faced with a rule with multiple premises and leftover threading, generate the inclusion evidence (Lemma 18) and use it to split up the witness of consumption equivalence (Lemma 17:3) and distribute it appropriately in the induction hypotheses. ◀

▶ **Example 20.** Coming back to the typing derivation for the de Bruijn index 1 in Example 8, we can use the Framing theorem to transport the proof that $\Gamma \cdot \mathtt{f}_\tau \cdot \mathtt{f}_\sigma \vdash_v \mathtt{s}\,\mathtt{z} \in \tau \boxtimes \Gamma \cdot \mathtt{s}_\tau \cdot \mathtt{f}_\sigma$ to a proof that $\Delta \cdot \mathtt{f}_\tau \cdot \mathtt{s}_\sigma \vdash_v \mathtt{s}\,\mathtt{z} \in \tau \boxtimes \Delta \cdot \mathtt{s}_\tau \cdot \mathtt{s}_\sigma$ for any Δ. Indeed, we can see that these two pairs of USAGE are consumption equivalent ($\Gamma - \Gamma \equiv \Delta - \Delta$ holds by induction):

$$\cfrac{\cfrac{\cfrac{\vdots}{\Gamma - \Gamma \equiv \Delta - \Delta}}{\Gamma \cdot \mathtt{f}_\tau - \Gamma \cdot \mathtt{s}_\tau \equiv \Delta \cdot \mathtt{f}_\tau - \Delta \cdot \mathtt{s}_\tau}}{\Gamma \cdot \mathtt{f}_\tau \cdot \mathtt{f}_\sigma - \Gamma \cdot \mathtt{s}_\tau \cdot \mathtt{f}_\sigma \equiv \Delta \cdot \mathtt{f}_\tau \cdot \mathtt{s}_\sigma - \Delta \cdot \mathtt{s}_\tau \cdot \mathtt{s}_\sigma}$$

5 Weakening

It is perhaps surprising to find a notion of weakening for a linear calculus: the whole point of linearity is precisely to ensure that all the resources are used. However, when opting for a system based on consumption annotations it becomes necessary to be able to extend the context a term lives in. This will typically be used in the definition of parallel substitution to push the substitution under a binder. Linearity is guaranteed by ensuring that the inserted variables are left untouched by the term.

Weakening arises from a notion of inclusion. The appropriate type theoretical structure to describe these inclusions is well-known and called an Order Preserving Embedding [15, 4]. Unlike a simple function witnessing the inclusion of its domain into its codomain, the restriction brought by order preserving embeddings guarantees that contraction is simply not possible which is crucial in a linear setting.

▶ **Definition 21.** An **Order Preserving Embedding** (OPE) is an inductive family. Its constructors (dubbed “moves” in this paper) describe a strategy to realise the promise of an injective embedding which respects the order induced by the de Bruijn indices. We start with an example in Figure 9 before giving, in Figure 10, the formal definition of OPEs for NAT, CONTEXT and USAGE.

In the following example, we prove that the source context $\gamma \cdot \tau$ can be safely embedded into the target one $\gamma \cdot \sigma \cdot \tau \cdot \nu$, written $\gamma \cdot \tau \leq \gamma \cdot \sigma \cdot \tau \cdot \nu$. This example proof uses all three of the moves the inductive definition of OPEs offers: $\mathtt{insert}_\alpha$ which introduces a new variable of type α, `copy` which embeds the source context's top variable, and `done` which simply copies the source context. Because we read strategies left-to-right and it is easier to see how they act if contexts are also presented left-to-right, we temporarily switch to *cons*-style (i.e. σ, γ) instead of the *snoc*-style (i.e. $\gamma \cdot \sigma$) used in the rest of this paper.

▶ **Example 22.** An Order Preserving Embedding.

Now that we have seen an example, we can focus on the formal definition. We give the definition of OPE for NAT, CONTEXT and USAGE all side by side in one table: the first column lists the names of the constructors associated to each move whilst the other ones

OPE	`insert`$_\nu$	`copy`	`insert`$_\sigma$	`done`
source		τ ,		γ
target	ν ,	τ ,	σ ,	γ

Figure 9 Example: proof that $\gamma \cdot \tau \leq \gamma \cdot \sigma \cdot \tau \cdot \nu$.

	NAT	CONTEXT	USAGE
`done`	$\dfrac{}{k \leq k}$	$\dfrac{}{\gamma \leq \gamma}$	$\dfrac{}{\Gamma \leq \Gamma}$
`copy`	$\dfrac{k \leq l}{1{+}k \leq 1{+}l}$	$\dfrac{\gamma \leq \delta}{\gamma \cdot \sigma \leq \delta \cdot \sigma}$	$\dfrac{\Gamma \leq \Delta \qquad S : \mathrm{USAGE}_\sigma}{\Gamma \cdot S \leq \Delta \cdot S}$
`insert`	$\dfrac{k \leq l}{k \leq 1{+}l}$	$\dfrac{\gamma \leq \delta}{\gamma \leq \delta \cdot \sigma}$	$\dfrac{\Gamma \leq \Delta \qquad S : \mathrm{USAGE}_\sigma}{\Gamma \leq \Delta \cdot S}$

Figure 10 Order Preserving Embeddings for NAT, CONTEXT and USAGE.

give their corresponding types for each category. It is worth noting that OPEs for CONTEXT are indexed over the ones for NAT and the OPEs for USAGE are indexed by both. The latter definitions are effectively algebraic *ornaments* [16, 28] over the previous ones, that is to say they have the same structure only storing additional information.

- The first row defines the move `done`. It is the strategy corresponding to the trivial embedding of a set into itself by the identity function and serves as a base case.
- The second row corresponds to the `copy` move which extends an existing embedding by copying the current 0th variable from source to target. The corresponding cases for CONTEXTs and USAGE are purely structural: no additional content is required to be able to perform a `copy` move.
- The third row describes the move `insert` which introduces an extra variable in the target set. This is the move used to extend an existing context, i.e. to weaken it. In this case, it is paramount that the OPE for CONTEXTs should take a type σ as an extra argument (it will be the type of the newly introduced variable) whilst the OPE for USAGE takes a USAGE$_\sigma$ (it will be the usage associated to that newly introduced variable of type σ).

Now that the structure of these OPEs is clear, we have to introduce a caveat regarding this description: the CONTEXT and USAGE case are a bit special. They do not in fact mention the source and target sets in their indices. This is a feature: when weakening a typing relation, the OPE for USAGE will be applied simultaneously to the input *and* the output USAGE which, although of a similar structure because of their shared CONTEXT index, will be different.

▶ **Definition 23.** The **semantics of an OPE** is defined by induction over the proof object. We use the overloaded function name ope($\cdot$) for it. They behave as the simplified view given in Figure 10 where γ / Γ is seen as the input, σ / S the additional information stored into the proof object and δ / Δ the output.

We leave out the definition of weakening for raw terms which is the standard definition for the untyped λ-calculus. It proves that given $k \leq l$ we can turn an INFER$_k$ (respectively CHECK$_k$) into an INFER$_l$ (respectively CHECK$_l$). It is given by a simple structural induction on the terms themselves, using `copy` to go under binders.

▶ **Definition 24.** A Typing Relation $\mathcal{T}$ for a NAT-indexed family T such that we have a function $\texttt{weak}_T$ transporting proofs that $k \leq l$ to functions $T_k \to T_l$ is said to have the **Weakening Property** if for all k, l in NAT, o a proof that $k \leq l$, O a proof that OPE(o) and $\mathcal{O}$ a proof that OPE(O) then for all γ a CONTEXT$_k$, Γ and Δ two USAGE$_\gamma$, t an element of T_k and σ a TYPE, if $\mathcal{T}_k(\Gamma, t, \sigma, \Delta)$ holds true then we also have $\mathcal{T}_l(\mathrm{ope}(\mathcal{O}, \Gamma), \texttt{weak}_T(o, t), \sigma, \mathrm{ope}(\mathcal{O}, \Delta))$.

▶ **Theorem 25.** *The Typing Relation for VAR has the Weakening Property. So do the Typing Relations for INFER and CHECK.*

Proof. The proof for VAR is by induction on the typing derivation. The statements for INFER and CHECK are proved by mutual structural inductions on the respective typing derivations. Using the `copy` constructor of OPEs is crucial to be able to go under binders. ◀

Unlike the framing property, this theorem is not purely about the type system: the term is indeed modified between the premise and the conclusion. Now that we know that weakening is compatible with the typing relations, let us study substitution.

6 Substituting

Stability of the typing relations under substitution guarantees that the untyped evaluation of programs will yield results which have the same type as well as preserve the linearity constraints. The notion of leftovers naturally extends to substitutions: the terms meant to be substituted for the variables in context which are not used by a term will not be used when pushing the substitution onto this term. They will therefore have to be returned as leftovers.

Because of this rather unusual behaviour for substitution, picking the right type-theoretical representation for the environment carrying the values to be substituted in is a bit subtle. Indeed, relying on the usual combination of weakening and crafting a fresh variable when going under a binder becomes problematic. The leftovers returned by the induction hypothesis would then live in an extended context and quite a lot of effort would be needed to downcast them back to the smaller context they started in. The solution is to have an explicit constructor for "going under a binder" which can be simply peeled off on the way out of a binder. The values are still weakened to fit in the extended context they end up in but that happens at the point of use (i.e. when they are being looked up to replace a variable) instead of when pushing the substitution under a binder.

▶ **Definition 26.** The environment ENV used to define substitution for raw terms is indexed by two NATs k and l where k is the source's scope and l is the target's scope. There are three constructors: one for the empty environment ($[]$), one for going under a binder ($\cdot \bullet \mathtt{v}$) and one to extend an environment with an INFER$_l$.

$$\frac{k, l : \mathrm{NAT}}{\mathrm{ENV}(k, l) : \mathit{Set}} \qquad \frac{}{[] : \mathrm{ENV}(0, l)} \qquad \frac{\rho : \mathrm{ENV}(k, l)}{\rho \bullet \mathtt{v} : \mathrm{ENV}(1{+}k, 1{+}l)} \qquad \frac{\rho : \mathrm{ENV}(k, l) \qquad t : \mathrm{INFER}_l}{\rho \cdot t : \mathrm{ENV}(1{+}k, l)}$$

Environment are carrying INFER elements because, being in the same syntactical class as VARs, they can be substituted for them without any issue. We now state the substitution lemma on untyped terms because it is, unlike the one for weakening, non-standard by way of our definition of environments.

▶ **Lemma 27.** *Given a VAR$_k$ v and an ENV(k, l) ρ, we can look up the INFER$_l$ associated to v in ρ.*

Proof. The proof goes by induction on v and case analysis on ρ. If the variable we look up has been introduced by a binder we went under using the constructor $\cdot$ •v then we return it immediately. Otherwise we get our hands on a term which we may need to weaken. This corresponds to the following functional specification (the practical implementation can be distinct to avoid retraversing the term once for every single binder we went under).

$$\begin{array}{ll}
\texttt{var z} & [\rho \bullet \mathtt{v}] = \mathtt{z} \\
\texttt{var z} & [\rho \cdot t] = t \\
\texttt{var}\ (\mathtt{s}\,v) & [\rho \bullet \mathtt{v}] = \mathtt{lift}(\mathtt{var}\ v\ [\rho]) \\
\texttt{var}\ (\mathtt{s}\,v) & [\rho \cdot t] = \mathtt{var}\ v\ [\rho]
\end{array}$$

We assume that `lift` is an instance of weakening defined in the previous section which takes a term in a scope of size k and returns the same term in scope $1+k$. ◀

▶ **Lemma 28.** *Raw terms are stable under substitutions: for all k and l, given t a term INFER_k (resp. CHECK_k) and ρ an environment $\mathrm{ENV}(k,l)$, we can apply the substitution ρ to t and obtain an INFER_l (resp. CHECK_l).*

Proof. By mutual induction on the raw terms. The traversals are purely structural except when going under binders where the constructor ($\cdot$ •v) is used to extend the ENV appropriately. The prototypical case of a binder is the `lam` one, and its functional specification is: $(\mathtt{lam}\ t)\ [\rho] = \mathtt{lam}\ (t\ [\rho \bullet \mathtt{v}])$. ◀

▶ **Definition 29.** The **environments** used when proving that Typing Relations are stable under substitution follow closely the ones for raw terms. $\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2$ is a typing relation with input usages Θ_1 and output Θ_2 for the raw substitution ρ targeting the fresh variables in Γ. The typing for the empty environment has the same input and output usages annotation. Formally:

$$\dfrac{\begin{array}{lll}\theta : \mathrm{CONTEXT}_l & & \\ \Theta_1 : \mathrm{USAGE}_\theta & & \\ \gamma : \mathrm{CONTEXT}_k & \rho : \mathrm{ENV}(k,l) & \Theta_2 : \mathrm{USAGE}_\theta \\ \Gamma : \mathrm{USAGE}_\gamma & & \end{array}}{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2 : \mathit{Set}} \qquad \dfrac{}{\Theta_1 \vdash_e [] \ni [] \boxtimes \Theta_1}$$

For **f**resh variables in Γ, there are two cases depending on whether they have been introduced by going under a binder or not. If it is not the case then the typing environment carries around a typing derivation for the term t meant to be substituted for this variable. Otherwise, it does not carry anything extra but tracks in its input / output usages annotation the fact that the variable has been consumed.

$$\dfrac{\Theta_1 \vdash t \in \sigma \boxtimes \Theta_2 \qquad \Theta_2 \vdash_e \Gamma \ni \rho \boxtimes \Theta_3}{\Theta_1 \vdash_e \Gamma \cdot \mathtt{f}_\sigma \ni \rho \cdot t \boxtimes \Theta_3} \qquad \dfrac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \cdot \mathtt{f}_\sigma \vdash_e \Gamma \cdot \mathtt{f}_\sigma \ni \rho \bullet \mathtt{v} \boxtimes \Theta_2 \cdot \mathtt{s}_\sigma}$$

For **s**tale variables, there are two cases too. They are however a bit more similar: none of them carry around an extra typing derivation. The main difference is in the shape of the input and output context: in the case for the “going under a binder” constructor, they are clearly enriched with an extra (now consumed) variable whereas it is not the case for the normal environment extension.

$$\dfrac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \vdash_e \Gamma \cdot \mathtt{s}_\sigma \ni \rho \cdot t \boxtimes \Theta_2} \qquad \dfrac{\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_2}{\Theta_1 \cdot \mathtt{s}_\sigma \vdash_e \Gamma \cdot \mathtt{s}_\sigma \ni \rho \bullet \mathtt{v} \boxtimes \Theta_2 \cdot \mathtt{s}_\sigma}$$

▶ **Definition 30.** A Typing Relation $\mathcal{T}$. for a NAT-indexed family T equipped with a function $\texttt{subst}_T$ which for all NATs k, l, given an element T_k and an ENV(k, l) returns an element T_l is said to be **stable under substitution** if for all NATs k and l, γ a CONTEXT$_k$, Γ and Δ two USAGE$_\gamma$, t an element of T_k, σ a Type, ρ an ENV(k, l), θ a CONTEXT$_l$ and Θ_1 and Θ_3 two USAGE$_\theta$ such that $\mathcal{T}_k(\Gamma, t, \sigma, \Delta)$ and $\Theta_1 \vdash_e \Gamma \ni \rho \boxtimes \Theta_3$ holds then there exists a Θ_2 of type USAGE$_\theta$ such that $\mathcal{T}_l(\Theta_1, \texttt{subst}_T(t, \rho), \sigma, \Theta_2)$ and $\Theta_2 \vdash_e \Delta \ni \rho \boxtimes \Theta_3$.

▶ **Theorem 31.** *The Typing Relations for INFER and CHECK are stable under substitution.*

Proof. The proof by mutual structural induction on the typing derivations relies heavily on the fact that these Typing Relations enjoy the framing property in order to adjust the USAGE annotations. ◀

7 Functionality

In the next section we will prove that type-checking and type-inference are decidable. In the cases where the check fails, we have to prove that any purported typing derivation would lead to a contradiction. These arguments all follow a similar pattern: assuming that a typing derivation exists, we use inversion lemmas to obtain results in direct contradiction to the observations we have made. These inversion lemmas often rely on the fact that the typing relations are functional.

Although we did highlight that some of our relations' indices are meant to be seen as inputs whilst others are supposed to be outputs, we have not yet made this relationship formal because this fact was seldom used in the proofs so far. Functionality can be expressed by saying that given a typing relation, if two typing derivations exist for some fixed arguments (seen as inputs) then the other arguments (seen as outputs) are equal to each other.

▶ **Definition 32.** We say that a relation R of type $\prod(ri : RI).II \to O(ri) \to Set$ is **functional** if for all $r$elevant $i$nputs ri, all pairs of $i$rrelevant $i$nputs ii_1 and ii_2 and for all pairs of $o$utputs o_1 and o_2, if both $R(ri, ii_1, o_1)$ and $R(ri, ii_2, o_2)$ hold then $o_1 \equiv o_2$.

▶ **Lemma 33.** *The Typing Relations for VAR and INFER are functional when seen as relations with relevant inputs the context and the scrutinee (either a VAR or an INFER), irrelevant inputs their USAGE annotation and outputs the inferred TYPEs.*

▶ **Lemma 34.** *The Typing Relations for VAR, INFER, CHECK and ENV are functional when seen as relations with relevant inputs all of their arguments except for one of the USAGE annotation or the other. This means that given a USAGE annotation (whether the input one or the output one) and the rest of the arguments, the other USAGE annotation is uniquely determined.*

8 Typechecking

▶ **Theorem 35** (Decidability of Typechecking). ***Type-inference** for INFER and **Type-checking** for CHECK are decidable. In other words, given a NAT k, γ a CONTEXT$_k$ and Γ a USAGE$_\gamma$,*

1. *for all INFER$_k$ t, we can decide if there is a TYPE σ and Δ a USAGE$_\gamma$ such that $\Gamma \vdash t \in \sigma \boxtimes \Delta$*
2. *for all TYPE σ and CHECK$_k$ t, we can decide if there is Δ a USAGE$_\gamma$ such that $\Gamma \vdash \sigma \ni t \boxtimes \Delta$.*

Proof. The proof proceeds by mutual induction on the raw terms, using inversion lemmas to dismiss the impossible cases, using auxiliary lemmas showing that typechecking of VARs and PATTERNs also is decidable and relies heavily on the functionality of the various relations involved. ◀

One of the benefits of having a formal proof of a theorem in Agda is that the theorem actually has computational content and may be run: the proof is a decision procedure.

▶ **Example 36.** We can for instance check that the search procedure succeeds in finding the `swapTyped` derivation we had written down as Example 13. Because σ and τ are abstract in the following snippet, the equality test checking that σ is equal to itself and so is τ does not reduce and we need to rewrite by the proof `eq-diag` that the equality test always succeeds in this kind of situation:

```
swapChecked : ∀ σ τ → check [] ((σ ⊗ τ) ⊸ (τ ⊗ σ)) swap
                    ≡ yes ([] , swapTyped)
swapChecked σ τ rewrite eq-diag τ | eq-diag σ = refl
```

9 Equivalence to ILL

We have now demonstrated that the USAGE-based formulation of linear logic as a type system is amenable to mechanisation without putting an unreasonable burden on the user. Indeed, the system's important properties can all be dealt with by structural induction and the user still retains the ability to write simple λ-terms which are not cluttered with structural rules.

However this presentation departs quite a lot from more traditional formulations of intuitionistic linear logic. This naturally raises the question of its correctness. In this section we recall a typical presentation of Intuitionistic Linear Logic using a Sequent Calculus, representing the multiset of assumptions as a list.

9.1 A Sequent Calculus for Intuitionistic Linear Logic

The definition of this calculus is directly taken from the Linear Logic Wiki [26] whose notations we follow to the letter. The interested reader will find more details in for instance Troelstra's lectures [35]. In the following figure, γ, δ, and θ are context variables while σ, τ , and ν range over types. We overload the comma to mean both consing a single type to the front of a list and appending two lists, as is customary.

Our only departure from the traditional presentation is the *mix* rule which is an artefact of our encoding multisets as lists. It allows the user to pick any interleaving θ of two lists γ and δ. This notion of interleaving is formalised by the following three-place relation.

▶ **Definition 37.** The **interleaving** relation is defined by three constructors: [] declares that interleaving two empty lists yields the empty-list whilst $\cdot ,_l \cdot$ (and $\cdot ,_r \cdot$ respectively) picks the head of the list on the left (the right respectively) as the head of the interleaving and the tail as the result of interleaving the rest.

$$\frac{\gamma, \delta, \theta : \mathrm{LIST}\, a}{\gamma, \delta \cong \theta : \mathit{Set}} \qquad \frac{}{[] : [], [] \cong []} \qquad \frac{p : \gamma, \delta \cong \theta}{\sigma ,_l p : (\sigma, \gamma), \delta \cong (\sigma, \theta)} \qquad \frac{p : \gamma, \delta \cong \theta}{\sigma ,_r p : \gamma, (\sigma, \delta) \cong (\sigma, \theta)}$$

Now that we have our definition of the usual representation of Intuitionistic Linear Logic (ILL), we are left with proving that the linear typing relation we have defined is both sound and complete with respect to that logic.

$$\frac{}{\sigma \vdash \sigma}\,ax \qquad \frac{\gamma \vdash \sigma \qquad \sigma, \delta \vdash \tau}{\gamma, \delta \vdash \tau}\,cut \qquad \frac{\gamma \vdash \sigma \qquad \delta \vdash \tau}{\gamma, \delta \vdash \sigma \otimes \tau}\,\otimes^R \qquad \frac{\tau, \sigma, \gamma \vdash \nu}{\sigma \otimes \tau, \gamma \vdash}\,\otimes^L \qquad \frac{}{\vdash \mathbb{1}}\,1^R$$

$$\frac{\gamma \vdash \sigma}{\mathbb{1}, \gamma \vdash \sigma}\,1^L \qquad \frac{}{\mathbb{0}, \gamma \vdash \sigma}\,0^L \qquad \frac{\sigma, \gamma \vdash \tau}{\gamma \vdash \sigma \multimap \tau}\,\multimap^R \qquad \frac{\gamma \vdash \sigma \qquad \tau, \delta \vdash \nu}{(\sigma \multimap \tau), \gamma, \delta \vdash \nu}\,\multimap^L$$

$$\frac{\gamma \vdash \sigma \qquad \gamma \vdash \tau}{\gamma \vdash \sigma \& \tau}\,\&^R \qquad \frac{\sigma, \gamma \vdash \nu}{\sigma \& \tau, \gamma \vdash \nu}\,\&^L_1 \qquad \frac{\tau, \gamma \vdash \nu}{\sigma \& \tau, \gamma \vdash \nu}\,\&^L_2 \qquad \frac{\gamma \vdash \sigma}{\gamma \vdash \sigma \oplus \tau}\,\oplus^R_1$$

$$\frac{\gamma \vdash \tau}{\gamma \vdash \sigma \oplus \tau}\,\oplus^R_2 \qquad \frac{\sigma, \gamma \vdash \nu \qquad \tau, \gamma \vdash \nu}{\sigma \oplus \tau, \gamma \vdash \nu}\,\oplus^L \qquad \frac{\gamma, \delta \vdash \sigma \qquad \gamma, \delta \cong \theta}{\theta \vdash \sigma}\,mix$$

Figure 11 Sequent Calculus for Intuitionistic Linear Logic.

9.2 Soundness

We start with the easiest part of the proof: soundness. This means that from a typing derivation, we can derive a proof in ILL of what is essentially the same statement. That is to say that if a term is said to have type σ in a fully fresh context γ and proceeds to consume all of the resources in that context during the typing derivation then it corresponds to a proof of $\gamma \vdash \sigma$ in ILL.

This statement needs to be generalised to be proven. Indeed, even if we start with a context full of available resources, at the first split we encounter (e.g. a tensor introduction or a function application), it won't be the case anymore in one of the sub-terms. To state this more general formulation, we need to introduce a new notion: used assumptions.

▶ **Definition 38.** The list of **used** assumptions in a proof $\Gamma \subseteq \Delta$ in the consumption partial order is the list of types which have turned from fresh in Γ to stale in Δ. The used$(\cdot)$ function is defined by recursion over the proof that $\Gamma \subseteq \Delta$.

▶ **Definition 39.** A Typing Relation $\mathcal{T}$ for terms T is said to be **sound** with respect to ILL if, for k a NAT, γ a CONTEXT$_k$, Γ and Δ two USAGE$_\gamma$, t a term T_k and σ a type, from the typing derivation $\mathcal{T}(\Gamma,\ \mathrm{t},\ \sigma,\ \Delta)$ and p a proof that $\Gamma \subseteq \Delta$ we can derive used$(p) \vdash \sigma$.

▶ Remark. The consumption lemma 18 guarantees that such a proof $\Gamma \subseteq \Delta$ always exists whenever the typing relation is either the one for VAR, INFER or CHECK.

Before we can prove the soudness theorem, we need two auxiliary lemmas allowing us to handle the mismatch we may have between the way the used assumptions of a derivation are computed and the way the ones for its subderivations are obtained.

▶ **Lemma 40.** *Given k a NAT, γ a CONTEXT$_k$ and Γ, Δ, and θ three USAGE$_\gamma$, we have:*

1. *if p and q are proofs that $\Gamma \subseteq \Delta$ then used(p) = used(q).*
2. *if p is a proof that $\Gamma \subseteq \Delta$, q that $\Delta \subseteq \theta$ and pq that $\Gamma \subseteq \theta$ then used(pq) is an interleaving of used(p) and used(q).*

The relation validating patterns is not a typing relation and as such it needs to be handled separately. This can be done by defining a procedure elaborating patterns away by showing that whenever $\sigma \ni p \rightsquigarrow \gamma$, it is morally acceptable to replace σ on the left by γ. Which gives us the following *cut*-like admissible rule:

▶ **Lemma 41** (Elaboration of Let-bindings). *Provided k a NAT, p a PATTERN$_k$, σ a TYPE and γ a CONTEXT$_k$ such that $\sigma \ni p \rightsquigarrow \gamma$, we have that for all δ and θ two LIST TYPE and τ a TYPE, if $\delta \vdash \sigma$ and $\gamma, \theta \vdash \tau$ then $\delta, \theta \vdash \tau$.*

We now have all the pieces to prove the soundness of our typing relations.

▶ **Theorem 42** (Soundness). *The typing relations for VAR, INFER and CHECK are all sound.*

Proof. The proof is by mutual induction on the typing derivations. ILL's right rules are in direct correspondence with our introduction rules. The eliminators in our languages are translated by using ILL's *cut* together with left rules. The *mix* rule is crucial to rewrite the derivations' contexts on the fly. ◀

9.3 Completeness

Completeness is a trickier thing to prove: given a derivation in the traditional sequent calculus, we need to build a corresponding term and its typing derivation. However, unlike the soundness one it does not give us any insight as to what the meaning of USAGE and typing derivations is. So we only state the result and give an idea of the proof.

▶ **Theorem 43** (Completeness). *Given γ a LIST TYPE and σ a type, from a proof $\gamma \vdash \sigma$ we can derive an INFER t and a proof that $\mathbf{f}_\gamma \vdash t \in \sigma \boxtimes \mathbf{s}_\gamma$.*

Proof. The proof is by induction over the derivation in ILL. The right rules match our introduction rules well enough that they do not pose any issue. Weakening lets us extend the USAGE of the sequents obtained by induction hypothesis in the case of multi-premises rules. Left rules are systematically translated as `cut` s.

Finally, the hardest rule to handle is the `mix` rule which reorganises the context. It is handled by a technical lemma which we have left out of this paper. Informally: it states that if the input and output USAGE of a typing derivation are obtained by the same interleaving of two distinct pairs of USAGE, then for any other interleaving we can find a term and a typing derivation for that term. ◀

10 Related Work

Benton, Bierman, de Paiva, and Hyland [8] did devise a term assignment system for Intuitionistic Linear Logic which was stable under substitution. Their system focuses on multiplicative linear logic only when ours also encompasses additive connectives *but* it gives a thorough treatment of the *!* modality. This is still an open problem for us because we do not want to have the explicit handling of *!*'s weakening, contraction, dereliction, and promotion rules pollute the raw terms.

Rand, Paykin and Zdancewic's work on modelling quantum circuits in Coq [34] necessarily includes a treatment of linearity as qbits cannot be duplicated. And because it is mechanised, they have to deal with the representation of contexts. Their focus is mostly on the quantum aspect and they are happy relying on Coq's scripting capabilities to cope with the extensional presentation.

Bob Atkey and James Wood [6] have been experimenting with using a deep embedding of a linear lambda calculus in Agda as a way to certify common algorithms. Being able to encode insertion sort as a term in this deep embedding is indeed sufficient to conclude that the output of the algorithm is a permutation of its input. They use on-the-fly re-ordering of contexts via explicit permutation proofs.

Polakow faced with the task of embedding a linear λ-calculus in Haskell [33] used a typed-tagless approach [23] and tried to get as much automation from typeclass resolution as possible. Seeing Haskell's typeclass resolution mechanism as a Prolog-style proof search engine, he opted for a relational description and thus an input-output presentation. This system can handle multiplicatives, additives and is even extended to a Dual Intuitionistic Linear Logic [7] to accomodate for values which can be duplicated. Focusing on the applications, it is not proven to be stable under substitution or that the typechecking process will always succeed.

The proof search community has been confronted with the inefficiency of randomly splitting up the multiset of assumption when applying a tensor-introduction rule. In an effort to combat this non-determinism, they have introduced alternative sequent calculi returning leftovers [14, 37]. However because they do not have to type a term living in a given context, they do not care about the structure of the context of assumptions: it is still modelled as a multiset.

We have already mentioned McBride's work [29] on (as a first approximation: the setup is actually more general) a type theory with a *dependent linear* function space as a very important source of inspiration. In that context it is indeed crucial to retain the ability to talk about a resource even if it has already been consumed. E.g. a function taking a boolean and deciding whether it is equal to `tt` or `ff` will have a type mentioning the function's argument twice. But in a lawful manner: $(x : \text{BOOL}) \multimap (x \equiv \texttt{tt}) \vee (x \equiv \texttt{ff})$. This leads to the need for a context *shared* across all subterms and consumption annotations ensuring that the linear resources are never used more than once.

Finally, we can find a very concrete motivation for a predicate similar to our USAGE in Robbert Krebbers' thesis [24]. In section 2.5.9, he describes one source of undefined behaviours in the C standard: the execution order of expressions is unspecified thus leaving the implementers with absolute freedom to pick any order they like if that yields better performances. To make their life simpler, the standard specifies that no object should be modified more than once during the execution of an expression. In order to enforce this invariant, Krebbers' memory model is enriched with extra information:

> [E]ach bit in memory carries a permission that is set to a special locked permission when a store has been performed. The memory model prohibits any access (read or store) to objects with locked permissions. At the next sequence point, the permissions of locked objects are changed back into their original permission, making future accesses possible again.

11 Conclusion

We have shown that taking seriously the view of linear logic as a logic of resource consumption leads, in type theory, to a well-behaved presentation of the corresponding type system for the lambda-calculus. The framing property claims that the state of irrelevant resources does not matter, stability under weakening shows that one may even add extra irrelevant assumptions to the context and they will be ignored whilst stability under substitution guarantees subject reduction with respect to the usual small step semantics of the lambda calculus. Finally, the decidability of type checking makes it possible to envision a user-facing language based on raw terms and top-level type annotations where the machine does the heavy lifting of checking that all the invariants are met whilst producing a certified-correct witness of typability.

Avenues for future work include a treatment of an *affine* logic where the type of substitution will have to be be different because of the ability to throw away resources without using them. Our long term goal is to have a formal specification of a calculus for Probabilistic and Bayesian Reasoning similar to the affine one described by Adams and Jacobs [2]. Another interesting question is whether these resource annotations can be used to develop a fully formalised proof search procedure for intuitionistic linear logic. The author and McBride have made an effort in such a direction [3] by designing a sound and complete search procedure for a fragment of intuitionistic linear logic with type constructors tensor and with. Its extension to lolipop is currently an open question.

References

1 Peter Achten, John Van Groningen, and Rinus Plasmeijer. High level specification of I/O in functional languages. In *Functional Programming, Glasgow 1992*, pages 1–17. Springer, 1993.

2 Robin Adams and Bart Jacobs. A Type Theory for Probabilistic and Bayesian Reasoning, 2015. URL: `http://arxiv.org/abs/1511.09230`.

3 Guillaume Allais and Conor McBride. Certified Proof Search for Intuitionistic Linear Logic, 2015. URL: `http://gallais.github.io/proof-search-ILLWiL/`.

4 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In *Category Theory and Computer Science*, pages 182–199. Springer, 1995.

5 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In *Computer Science Logic*, pages 453–468. Springer, 1999.

6 Robert Atkey and James Wood. Sorting Types – Permutation via Linearity. `https://github.com/bobatkey/sorting-types`, 2013. Retrieved on 2017-11-06.

7 Andrew Barber and Gordon Plotkin. *Dual intuitionistic linear logic.* University of Edinburgh, Department of Computer Science, Laboratory for Foundations of Computer Science, 1996.

8 Nick Benton, Gavin Bierman, Valeria De Paiva, and Martin Hyland. A term calculus for intuitionistic linear logic. *Typed Lambda Calculi and Applications*, pages 75–90, 1993.

9 Richard S. Bird and Ross Paterson. de Bruijn notation as a nested datatype. *Journal of Functional Programming*, 9(1):77–91, 1999.

10 Samuel Boutin. Using reflection to build efficient and certified decision procedures. In *Theoretical Aspects of Computer Software*, pages 515–529. Springer, 1997.

11 Edwin Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23(05):552–593, 2013.

12 Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for proofs and programs*, pages 115–129. Springer, 2003.

13 Edwin Brady and Matúš Tejiščák. Practical Erasure in Dependently Typed Languages.

14 Iliano Cervesato, Joshua S Hodas, and Frank Pfenning. Efficient resource management for linear logic proof search. In *International Workshop on Extensions of Logic Programming*, pages 67–81. Springer, 1996.

15 James Maitland Chapman. *Type checking and normalisation.* PhD thesis, University of Nottingham, 2009.

16 Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of Functional Programming*, 24(2-3):316–383, 2014.

17 Nils Danielsson. Bag Equivalence via a Proof-Relevant Membership Relation. *Interactive Theorem Proving*, pages 149–165, 2012.

18 Olivier Danvy. Type-Directed Partial Evaluation. In *Partial Evaluation*, pages 367–411. Springer, 1999.

19 Nicolaas Govert De Bruijn. Lambda Calculus Notation with Nameless Dummies. In *Indagationes Mathematicae (Proceedings)*, volume 75, pages 381–392. Elsevier, 1972.

20 The Rust Project Developers. The Rust Programming Language – Ownership. `https://doc.rust-lang.org/book/first-edition/ownership.html`, 2017. Retrieved on 2017-11-06.

21 Peter Dybjer. Inductive families. *Formal aspects of computing*, 6(4):440–465, 1994.

22 Jean-Yves Girard. Linear logic. *Theoretical computer science*, 50(1):1–101, 1987.

23 Oleg Kiselyov. Typed tagless final interpreters. In *Generic and Indexed Programming*, pages 130–174. Springer, 2012.

24 Robbert Krebbers. *The C standard formalized in Coq*. PhD thesis, Radboud University Nijmegen, 2015.

25 John Launchbury and Simon L Peyton Jones. Lazy functional state threads. In *ACM SIGPLAN Notices*, volume 29, pages 24–35. ACM, 1994.

26 Olivier Laurent and Laurent Regnier. Linear Logic Wiki – Intuitionistic Linear Logic. `http://llwiki.ens-lyon.fr/mediawiki/index.php/Intuitionistic_linear_logic`, 2009. Retrieved on 2017-11-06.

27 Pierre Letouzey. A new extraction for Coq. In *Types for proofs and programs*, pages 200–219. Springer, 2002.

28 Conor McBride. Ornamental algebras, algebraic ornaments. *Journal of functional programming*, 2010.

29 Conor McBride. I Got Plenty o'Nuttin'. In *A List of Successes That Can Change the World*, pages 207–233. Springer, 2016.

30 John C Mitchell. *Foundations for programming languages*, volume 1. MIT press, 1996.

31 Ulf Norell. Dependently typed programming in Agda. In *Advanced Functional Programming*, pages 230–266. Springer, 2009.

32 Benjamin C Pierce and David N Turner. Local type inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22(1):1–44, 2000.

33 Jeff Polakow. Embedding a full linear lambda calculus in Haskell. *ACM SIGPLAN Notices*, 50(12):177–188, 2016.

34 Robert Rand, Jennifer Paykin, and Steve Zdancewic. QWIRE Practice: Formal Verification of Quantum Circuits in Coq. In *Quantum Physics and Logic*, 2017.

35 Anne Sjerp Troelstra. *Lectures on linear logic*. Center for the Study of Language and Inf, 1991.

36 Stephanie Weirich, Justin Hsu, and Richard A Eisenberg. Towards dependently typed Haskell: System FC with kind equality. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*. Citeseer, 2013.

37 Michael Winiko and James Harland. Deterministic resource management for the linear logic programming language Lygon. Technical report, Technical Report 94/23, Melbourne University, 1994.

A Fully-expanded Typing Derivation for `swap`

$$\dfrac{\dfrac{\mathcal{E} \qquad \mathcal{P} \qquad \dfrac{\mathcal{L} \qquad \mathcal{R}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{f}_{\tau} \cdot \texttt{f}_{\sigma} \vdash \tau \otimes \sigma \ni \texttt{prd (neu (var (s z))) (neu (var z))} \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{s}_{\sigma}}}{[] \cdot \texttt{f}_{\sigma\otimes\tau} \vdash \tau \otimes \sigma \ni \begin{array}{l}\texttt{let (v , v) := var z in} \\ \texttt{prd (neu (var (s z))) (neu (var z))}\end{array} \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau}}}{[] \vdash (\sigma \otimes \tau) \multimap (\tau \otimes \sigma) \ni \texttt{swap} \boxtimes []}$$

$$\mathcal{E} = \dfrac{\dfrac{}{[] \cdot \texttt{f}_{\sigma\otimes\tau} \vdash_{v} \texttt{z} \in \sigma \otimes \tau \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau}}}{[] \cdot \texttt{f}_{\sigma\otimes\tau} \vdash \texttt{var z} \in \sigma \otimes \tau \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau}}$$

$$\mathcal{P} = \dfrac{\dfrac{}{\sigma \ni \texttt{v} \leadsto [] \cdot \sigma} \qquad \dfrac{}{\tau \ni \texttt{v} \leadsto [] \cdot \tau}}{\sigma \otimes \tau \ni \texttt{(v , v)} \leadsto [] \cdot \tau \cdot \sigma}$$

$$\mathcal{L} = \dfrac{\dfrac{\dfrac{\dfrac{}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{f}_{\tau} \vdash_{v} \texttt{z} \in \tau \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau}}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{f}_{\tau} \cdot \texttt{f}_{\sigma} \vdash_{v} \texttt{s z} \in \tau \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma}}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{f}_{\tau} \cdot \texttt{f}_{\sigma} \vdash \texttt{var (s z)} \in \tau \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma}}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{f}_{\tau} \cdot \texttt{f}_{\sigma} \vdash \tau \ni \texttt{neu (var (s z))} \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma}}$$

$$\mathcal{R} = \dfrac{\dfrac{\dfrac{}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma} \vdash_{v} \texttt{z} \in \sigma \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{s}_{\sigma}}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma} \vdash \texttt{var z} \in \sigma \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{s}_{\sigma}}}{[] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{f}_{\sigma} \vdash \sigma \ni \texttt{neu (var z)} \boxtimes [] \cdot \texttt{s}_{\sigma\otimes\tau} \cdot \texttt{s}_{\tau} \cdot \texttt{s}_{\sigma}}$$

Lower End of the Linial-Post Spectrum

Andrej Dudenhefner
Technical University of Dortmund, Dortmund, Germany
andrej.dudenhefner@cs.tu-dortmund.de

Jakob Rehof
Technical University of Dortmund, Dortmund, Germany
jakob.rehof@cs.tu-dortmund.de

Abstract

We show that recognizing axiomatizations of the Hilbert-style calculus containing only the axiom $a \to (b \to a)$ is undecidable (a reduction from the Post correspondence problem is formalized in the Lean theorem prover). Interestingly, the problem remains undecidable considering only axioms which, when seen as simple types, are principal for some λ-terms in β-normal form. This problem is closely related to type-based composition synthesis, i.e. finding a composition of given building blocks (typed terms) satisfying a desired specification (goal type).

Contrary to the above result, axiomatizations of the Hilbert-style calculus containing only the axiom $a \to (b \to b)$ are recognizable in linear time.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases combinatory logic, lambda calculus, type theory, simple types, inhabitation, principal type

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.2

Acknowledgements We thank the anonymous reviewers for their comprehensive and particularly constructive comments.

1 Introduction

The problem of decidability of a given Hilbert-style propositional calculus was posed by Tarski in 1946 and has subsequently been studied by several authors in various forms, beginning with the abstract published in 1949 by Linial and Post [13] in which the existence of an undecidable propositional calculus and undecidability of the problem of recognizing axiomatizations of classical propositional logic was stated (Linial-Post theorems). Zolin [20] provides a good overview of the history of this problem, and let us mention here only that Singletary [16] proved in 1974 that there exists a purely implicational propositional calculus which can represent any r.e. degree. Such a calculus can be seen as a combinatory logic [1, 11] in simple types [10, 2] with an undecidable inhabitation problem. In honor of the pioneers, we refer to the set of all Hilbert-style propositional calculi as the Linial-Post spectrum. In the present work we shed some light on the 'lower end' (seemingly very weak calculi) of the Linial-Post spectrum from the point of view of functional program synthesis. Our main result is that recognizing axiomatizations of the Hilbert-style calculus containing only the axiom $a \to (b \to a)$ (the type of the combinator $\mathbf{K}$ in combinatory logic) is undecidable. Moreover, we show that the problem remains undecidable considering only *principal* axioms, i.e. axioms that, seen as simple types, are principal for some λ-terms in β-normal form. In general, to recognize whether given axioms $\Delta = \{\sigma_1, \dots, \sigma_n\}$ axiomatize some calculus $\mathfrak{C}$ means to decide whether theorems in $\mathfrak{C}$ coincide with formulae derivable from Δ using the rules of substitution and modus ponens. In particular, if $\mathfrak{C}$ is the Hilbert-style calculus containing only the axiom $a \to (b \to a)$, then Δ axiomatizes $\mathfrak{C}$ iff $a \to (b \to a)$ is derivable from Δ and σ is derivable from $\{a \to (b \to a)\}$ for each $\sigma \in \Delta$. We show that the former problem, even if only principal axioms are considered, is undecidable while the latter, in general, is decidable in linear time[1]. If one is not interested in principal axioms, then the former result follows directly from the recent work by Bokov [6] which shows that problems of recognizing axiomatizations and completeness are undecidable for propositional calculi containing the axiom $a \to (b \to a)$.

The result presented here encompasses two motivating aspects which distinguish it from existing work (see [20] for an overview). First, similar to Bokov [6], we explore the lower end of the Linial-Post spectrum, whereas existing work focuses on classical [13, 19, 4] or superintuitionistic [12, 20, 5] calculi, often having rich type syntax, e.g. containing negation. In this work, we consider only implicational formulae and stay below $a \to (b \to a)$ in terms of derivability. This is arguably 'as low as you can get' because, as will be shown, axiomatizations of $a \to a$ (or even $a \to (b \to b)$) are recognizable in linear time. Second, we are interested in synthesis of functional programs from given building blocks. Following the same motivation as [15, 3], we want to utilize proof search (inhabitation in combinatory logics) to synthesize code by composition from a given collection of combinators. Specifically, provided simply typed λ-terms $M_1, \dots, M_n$ in β-normal form, we search for an applicative composition of the given terms that has some designated simple type σ. This is equivalent to proof search in the Hilbert-style calculus having axioms $\sigma_1, \dots, \sigma_n$ where σ_i is the principal type of M_i for $i = 1 \dots n$. It is a typical synthesis scenario, in which $M_1, \dots, M_n$ are library components exposing functionality specified by their corresponding principal types $\sigma_1, \dots, \sigma_n$. The synthesized composition is a functional program that uses the library to realize behavior specified by the type σ.

Our second motivation forces us to deviate from standard constructions pervading existing work. For example, considering axioms $a \to (a \to a)$ (testing equality of two arguments) or $(a \to b) \to b$ (encoding disjunction), there are no λ-terms in β-normal form having such axioms as their principal types. Therefore, such logical formulae could be considered an artificial and purely logical peculiarity from the point of view of program synthesis. Moreover, necessarily deviating from existing techniques (also using the Post correspondence problem instead of Post production systems as in [6]) we provide a novel and formalized[2] proof.

A noteworthy side effect when considering axioms that are tied to corresponding λ-terms via principality is an additional twist to the Curry-Howard isomorphism. We observe that the constructed proof that a formula is derivable (therefore logically solving the underlying problem) corresponds to a λ-term that actually solves the underlying problem computationally.

The paper is organized as follows. Section 2 recapitulates preliminary definitions (simply typed λ-calculus, simply typed combinatory logic, Hilbert-style calculi and the Post correspondence problem). Our main result on undecidability of recognizing axiomatizations of $a \to (b \to a)$ in shown in Section 3, which also contains linear time derivability from $a \to (b \to a)$. The proof is formalized[2] in the Lean theorem prover. Formalizations of key statements are referred to by `namespace.lemma`. Complementary, linear time decidability of recognizing axiomatizations of $a \to a$ (resp. $a \to (b \to b)$) are shown in Section 4 (resp. Section 5). We conclude the paper in Section 6 which also contains remarks on future work.

[1] As is well known, adding the axiom $(a \to (b \to c)) \to ((a \to b) \to (a \to c))$, the type of the combinator $\mathbf{S}$, results in intuitionistic implicational logic for which provability is Pspace-complete [18].

[2] `http://www-seal.cs.tu-dortmund.de/seal/downloads/research/TYPES17.zip`

2 Preliminaries

In this section we briefly assemble necessary prerequisites in order to discuss principal axiomatizations of implicational propositional calculi. For a survey on simply typed calculi along with corresponding (under the Curry-Howard isomorphism) implicational propositional calculi see [17].

2.1 Simply Typed Lambda Calculus

We denote *λ-terms* (cf. Definition 1) by M, N, L where *term variables* are denoted by x, y, z. As is usual, application associates to the left and binds stronger than abstraction.

▶ **Definition 1** (λ-Terms). $M, N, L ::= x \mid (\lambda x.M) \mid (M\ N)$

Simple types (cf. Definition 2) are denoted by σ, τ where *type atoms* (also called *type variables* in literature) are denoted by a, b, c and drawn from the denumerable set $\mathbb{A}$. As is usual, $\to$ associates to the right, i.e. $\sigma \to \tau \to \sigma = \sigma \to (\tau \to \sigma)$.

▶ **Definition 2** (Simple Types). $\mathbb{T} \ni \sigma, \tau ::= a \mid \sigma \to \tau$

A *type environment* Γ is a finite set of *type assumptions* $\{x_1 : \sigma_1, \dots, x_n : \sigma_n\}$ in which term variables occur at most once. We set

$$\mathrm{dom}(\Gamma) = \{x_1, \dots, x_n\}, \qquad |\Gamma| = \{\sigma_1, \dots, \sigma_n\}, \qquad \Gamma(x_i) = \sigma_i \text{ for } i = 1 \dots n$$

We write $\Gamma \cup \{x : \sigma\}$ as $\Gamma, x : \sigma$ if $x \notin \mathrm{dom}(\Gamma)$.

The rules (Var), ($\to$I) and ($\to$E) of the *simple type system* ($\vdash$) are given in the following Definition 3.

▶ **Definition 3** (Simply Typed λ-Calculus ($\vdash$)).

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma}\ (\mathrm{Var}) \qquad \frac{\Gamma, x : \sigma \vdash M : \tau}{\Gamma \vdash \lambda x.M : \sigma \to \tau}\ (\to\mathrm{I}) \qquad \frac{\Gamma \vdash M : \sigma \to \tau \qquad \Gamma \vdash N : \sigma}{\Gamma \vdash M\ N : \tau}\ (\to\mathrm{E})$$

A *substitution* $\zeta : \mathbb{A} \to \mathbb{T}$ is a mapping such that its *substitution domain* $\mathrm{dom}(\zeta) := \{a \in \mathbb{A} \mid \zeta(a) \neq a\}$ is finite. We lift substitutions homomorphically to types. We say σ is *unifiable* with τ, if there exists a substitution ζ such that $\zeta(\sigma) = \zeta(\tau)$. A *principal type* (cf. Definition 4) of a term is the most general type assignable to that term and is unique up to atom renaming.

▶ **Definition 4** (Principal Type). We say that τ is a *principal type* of M, if $\vdash M : \tau$ and for all types σ such that $\vdash M : \sigma$ there exists a substitution ζ such that $\zeta(\tau) = \sigma$.

2.2 Simply Typed Combinatory Logic

We call λ-terms without abstractions *combinatory terms* (cf. Definition 5), denoted by F, G, H.

▶ **Definition 5** (Combinatory Terms). $F, G, H ::= x \mid (F\ G)$

The *size* of a combinatory term is the number of leaves in its syntax tree (cf. Definition 6).

▶ **Definition 6** (Size). $\mathrm{size}(x) = 1$ and $\mathrm{size}(F\ G) = \mathrm{size}(F) + \mathrm{size}(G)$.

The rules (Ax) and ($\to$E) of the *simply typed combinatory logic* ($\vdash_{\mathcal{C}}$) are given in the following Definition 7.

▶ **Definition 7** (Simply Typed Combinatory Logic ($\vdash_{\mathcal{C}}$)).

$$\frac{\zeta \text{ is a substitution}}{\Gamma, x : \sigma \vdash_{\mathcal{C}} x : \zeta(\sigma)}\ (\text{Ax}) \qquad \frac{\Gamma \vdash_{\mathcal{C}} F : \sigma \to \tau \qquad \Gamma \vdash_{\mathcal{C}} G : \sigma}{\Gamma \vdash_{\mathcal{C}} F\ G : \tau}\ (\to\text{E})$$

Observe that the above definition is *relativized* to arbitrary bases Γ whereas 'simply typed combinatory logic' often refers to a fixed base containing the combinators **S** and **K** with their corresponding types. We will use relativization to inspect arbitrary bases allowing to go below **S** and **K** in term of derivability.

Naturally, combinatory terms are of shape $x\ F_1 \dots F_n$ for some $n \in \mathbb{N}$ and we have the following generation lemma (cf. `derivation.long_typability`).

▶ **Lemma 8** (Generation Lemma). *If $\Gamma \vdash_{\mathcal{C}} x\ F_1 \dots F_n : \tau$, then there exists a substitution ζ such that $\zeta(\Gamma(x)) = \sigma_1 \to \dots \to \sigma_n \to \tau$ for some $n \in \mathbb{N}$, $\sigma_1, \dots, \sigma_n$ and $\Gamma \vdash_{\mathcal{C}} F_i : \sigma_i$ holds for $i = 1 \dots n$.*

To mitigate extensive use of parentheses in combinatory terms we use the left-associative *pipe* metaoperator $\triangleright$ defined as $F \triangleright G = (G\ F)$. For example, $G_3\ (G_2\ (G_1\ F)) = F \triangleright G_1 \triangleright G_2 \triangleright G_3$.

2.3 Hilbert-Style Calculus

We identify propositional implicational *axioms* (sometimes called *formulae*) with simple types and denote finite sets of axioms by Δ. The rules (Ax) and ($\to$E) of the *Hilbert-style calculus* ($\vdash_{\mathcal{C}}$) are given in the following Definition 9.

▶ **Definition 9** (Hilbert-Style Calculus ($\vdash_{\mathcal{H}}$)).

$$\frac{\zeta \text{ is a substitution}}{\Delta, \sigma \vdash_{\mathcal{H}} \zeta(\sigma)}\ (\text{Ax}) \qquad \frac{\Delta \vdash_{\mathcal{H}} \sigma \to \tau \qquad \Delta \vdash_{\mathcal{H}} \sigma}{\Delta \vdash_{\mathcal{H}} \tau}\ (\to\text{E})$$

Again, ($\vdash_{\mathcal{H}}$) is relativized to arbitrary sets of axioms Δ. Observe that ($\vdash_{\mathcal{H}}$) and ($\vdash_{\mathcal{C}}$) are in direct Curry-Howard correspondence. The set of *derivable* formulae is denoted by $[\Delta]_{\mathcal{H}} = \{\tau \in \mathbb{T} \mid \Delta \vdash_{\mathcal{H}} \tau\}$.

We say Δ_1 *axiomatizes* $[\Delta_2]_{\mathcal{H}}$ if $[\Delta_1]_{\mathcal{H}} = [\Delta_2]_{\mathcal{H}}$. Clearly, $[\Delta_1]_{\mathcal{H}} = [\Delta_2]_{\mathcal{H}}$ iff $\Delta_1 \vdash_{\mathcal{H}} \tau$ for all $\tau \in \Delta_2$ and $\Delta_2 \vdash_{\mathcal{H}} \sigma$ for all $\sigma \in \Delta_1$. For brevity, we say Δ axiomatizes σ if $[\Delta]_{\mathcal{H}} = [\{\sigma\}]_{\mathcal{H}}$.

▶ **Example 10.** For $\Delta = \{a \to b \to a, (a \to b \to c) \to (a \to b) \to a \to c\}$ the set of formulae $[\Delta]_{\mathcal{H}}$ contains exactly the intuitionistic propositional implicational tautologies.

For $\Delta' = \Delta \cup \{((p \to q) \to p) \to p\}$ the set of formulae $[\Delta']_{\mathcal{H}}$ contains exactly the classical propositional implicational tautologies [20].

▶ **Definition 11.** An axiom σ is *principal* if there exists a λ-term M in β-normal form such that σ is the principal type of M in the simply typed λ-calculus.

Intuitively, axioms that are not principal, e.g. $a \to a \to a$, could in some contexts (for example, in synthesis) be considered 'artificial' since they have no 'naturally' associated realization. Principality of axioms is decidable [7] and, in fact, PSPACE-complete [9].

2.4 Post Correspondence Problem

The Post correspondence problem (PCP) is well-known for its undecidability ([14]).

▶ **Problem 12** (Post Correspondence Problem). *Given pairs of words $(v_1, w_1), \dots, (v_k, w_k)$ over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ such that $v_i \neq \epsilon \neq w_i$ for $i = 1 \dots k$ (where ϵ is the empty word) decide whether there exists an index sequence $i_1, \dots, i_n$ such that $v_{i_1} v_{i_2} \dots v_{i_n} = w_{i_1} w_{i_2} \dots w_{i_n}$.*

As an immediate consequence, the following slight restriction of the problem, where $v_i \neq w_i$, is also undecidable.

▶ **Corollary 13.** *Given pairs of words* $(v_1, w_1), \ldots, (v_k, w_k)$ *over the alphabet* $\{\mathtt{a}, \mathtt{b}\}$ *such that* $\epsilon \neq v_i \neq w_i \neq \epsilon$ *for* $i = 1 \ldots k$ *it is undecidable whether there exists an index sequence* $i_1, \ldots, i_n$ *such that* $v_1 v_{i_1} v_{i_2} \ldots v_{i_n} = w_1 w_{i_1} w_{i_2} \ldots w_{i_n}$.

We aim at showing undecidability of recognizing principal axiomatizations of the calculus $a \to b \to a$ by reduction from PCP. Usually, the Post correspondence problem is approached constructively, i.e. start with some given pair of words, then iteratively append corresponding suffixes, and finally test for equality. The approach taken in the present work is, in a sense, 'deconstructive'. In particular, we start from an arbitrary pair of equal words, then iteratively remove corresponding suffixes, and finally test whether the resulting pair is given. While the former approach requires an equality test for arbitrarily large structures as a final operation (the encoding of which appears problematic in terms of principal axioms), the final operation of the latter approach can be bounded. The following Definition 14 and Lemma 15 capture the outlined iterative deconstruction.

▶ **Definition 14.** Given a set $\mathsf{PCP} = \{(v_1, w_1), \ldots, (v_k, w_k)\}$ of pairs of words over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ we define for $n \geq 0$ the set PCP_n of pairs of words as follows

$$\mathsf{PCP}_0 = \{(v, v) \mid v \in \{\mathtt{a}, \mathtt{b}\}^*\} \qquad \mathsf{PCP}_{n+1} = \{(v, w) \mid \exists i \in \{1, \ldots, k\}.(vv_i, ww_i) \in \mathsf{PCP}_n\}$$

▶ **Lemma 15.** *Let* $n \geq 0$ *and* $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. *We have* $vv_{i_1} v_{i_2} \ldots v_{i_n} = ww_{i_1} w_{i_2} \ldots w_{i_n}$ *for some index sequence* $i_1, \ldots, i_n$ *iff* $(v, w) \in \mathsf{PCP}_n$.

Proof. Routine induction on n (cf. `pcp.pcp_set_iff_sync_word_pair`). ◀

In sum, it is undecidable whether the prefix (v_1, w_1) is in PCP_n for some $n \geq 0$.

▶ **Lemma 16.** *Given a set* $\mathsf{PCP} = \{(v_1, w_1), \ldots, (v_k, w_k)\}$ *of pairs of words over the alphabet* $\{\mathtt{a}, \mathtt{b}\}$ *such that* $\epsilon \neq v_i \neq w_i \neq \epsilon$ *for* $i = 1 \ldots k$ *it is undecidable whether there exists an* $n \geq 0$ *such that* $(v_1, w_1) \in \mathsf{PCP}_n$.

Proof. Immediate consequence of Corollary 13 and Lemma 15. ◀

3 Recognizing Axiomatizations of $a \to b \to a$

In this section we show that recognizing principal axiomatizations of the Hilbert-style calculus containing only the axiom $a \to b \to a$ is undecidable (cf. Theorem 17), which is our main result.

▶ **Theorem 17.** *Given principal axioms* $\sigma_1, \ldots, \sigma_n$ *such that* $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma_i$ *for* $i = 1 \ldots n$, *it is undecidable whether* $\{\sigma_1, \ldots, \sigma_n\} \vdash_{\mathcal{H}} a \to b \to a$.

▶ **Corollary 18.** *Given* λ*-terms* $M_1, \ldots, M_n$ *in* β*-normal form with principal types* $\sigma_1, \ldots, \sigma_n$ *in the simply typed* λ*-calculus such that* $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma_i$ *for* $i = 1 \ldots n$, *it is undecidable whether there is an applicative composition of* $M_1, \ldots, M_n$ *having the simple type* $a \to b \to a$.

In the context of type-based composition synthesis, the types $\sigma_1, \ldots, \sigma_n$ are natural specifications of associated terms $M_1, \ldots, M_n$ and $a \to b \to a$ is a goal specification. Deriving $a \to b \to a$ from $\sigma_1, \ldots, \sigma_n$ naturally corresponds to finding a composition of given terms satisfying the goal specification.

We prove Theorem 17 by reduction from the Post correspondence problem, specifically, the construction in Lemma 16. For that reason, we fix pairs $(v_1, w_1), \ldots, (v_k, w_k)$ of words over the alphabet $\{\mathtt{a}, \mathtt{b}\}$ such that $\epsilon \neq v_i \neq w_i \neq \epsilon$ for $i = 1 \ldots k$. Our goal is to construct principal axioms $\sigma_1, \ldots, \sigma_l$ such that $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma_i$ for $i = 1 \ldots l$ and $\{\sigma_1, \ldots, \sigma_l\} \vdash_{\mathcal{H}} a \to b \to a$ is equivalent to $(v_1, w_1) \in \mathsf{PCP}_n$ for some $n \geq 0$.

PCP Reduction

In this subsection we construct axioms $\sigma_1, \ldots, \sigma_l$ associated with the outlined reduction. We do not address principality which will be dealt with in the next subsection.

We need to represent words, pairs and suffixing. Let us fix a unique type atom $\bullet$. For a word $v \in \{\mathtt{a}, \mathtt{b}\}^*$ we define its representation as $[v] = \bullet \cdot v$ where the operation $\cdot$ is defined as

$$\sigma \cdot \epsilon = \sigma \qquad \sigma \cdot w\mathtt{a} = (\bullet \to \bullet) \to (\sigma \cdot w) \qquad \sigma \cdot w\mathtt{b} = (\bullet \to \bullet \to \bullet) \to (\sigma \cdot w)$$

We represent a pair of types σ, τ as

$$\langle \sigma, \tau \rangle = (\bullet \to \bullet \to \bullet) \to (\sigma \to \tau \to \bullet) \to (\bullet \to \sigma) \to (\bullet \to \tau) \to \bullet \to \bullet \to \bullet$$

As a side note, $[v]$ contains only $\bullet$ as atom. More importantly, we have $[v] \cdot w = [vw]$, and representations of two distinct words are not unifiable (cf. Lemma 19).

▶ **Lemma 19.** *Let $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. If $[v]$ and $[w]$ are unifiable, then $v = w$.*

Proof. Assuming $v \neq w$ we show that $[v]$ and $[w]$ are not unifiable by induction on the length of v (cf. `word.append_unique_encoding`). Wlog. v is not longer than w.

Case $v = \epsilon$: Clearly, $[v] = \bullet$ is not unifiable with $[w] = \sigma_1 \to \ldots \to \sigma_n \to \bullet$ with $n \geq 1$.

Case $v = v'\mathtt{a}$, $w = w'\mathtt{b}$ (resp. $v = v'\mathtt{b}$, $w = w'\mathtt{a}$): Let $\sigma = \bullet \to \bullet$ (resp. $\sigma = \bullet \to \bullet \to \bullet$) and $\tau = \bullet \to \bullet \to \bullet$ (resp. $\tau = \bullet \to \bullet$). Since σ is not unifiable with τ, we have that $[v] = \sigma \to [v']$ is not unifiable with $[w] = \tau \to [w']$.

Case $v = v'\mathtt{a}$, $w = w'\mathtt{a}$ (resp. $v = v'\mathtt{b}$, $w = w'\mathtt{b}$): By induction hypothesis $[v']$ is not unifiable with $[w']$. Therefore, $[v] = \sigma \to [v']$ is not unifiable with $[w] = \sigma \to [w']$, where $\sigma = \bullet \to \bullet$ (resp. $\sigma = \bullet \to \bullet \to \bullet$). ◀

Additionally, for any types σ, τ we have that $\langle \sigma, \tau \rangle$ is derivable from $a \to b \to a$ (cf. Lemma 21).

▶ **Lemma 20.** *Let σ, τ be types. If $\{a \to b \to a\} \vdash_{\mathcal{H}} \tau$, then $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma \to \tau$.*

Proof. Use $(\to\mathrm{E})$ with the premises $\{a \to b \to a\} \vdash_{\mathcal{H}} \tau \to \sigma \to \tau$ and $\{a \to b \to a\} \vdash_{\mathcal{H}} \tau$. ◀

▶ **Lemma 21.** *Let σ, τ be types. We have $\{a \to b \to a\} \vdash_{\mathcal{H}} \langle \sigma, \tau \rangle$.*

Proof. Iterative application of Lemma 20 starting with $\{a \to b \to a\} \vdash_{\mathcal{H}} \bullet \to \bullet \to \bullet$. ◀

Finally, we define a type environment Γ of $k + 2$ combinators typed by principal axioms

$$\Gamma = \{x : \langle a, a \rangle, z : \langle [v_1], [w_1] \rangle \to \bullet \to a \to \bullet\} \cup \{y_i : \langle a \cdot v_i, b \cdot w_i \rangle \to \langle a, b \rangle \mid 1 \leq i \leq k\}$$

Due to Lemma 21, each axiom in $|\Gamma|$ is derivable from $a \to b \to a$.

Having established all prerequisite definitions, we now proceed with our main reduction. The following Lemma 22 establishes a connection between elements $(v, w) \in \mathsf{PCP}_n$ and inhabitants of $\langle [v], [w] \rangle$.

▶ **Lemma 22.** *Let ζ be a substitution and let $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. If $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \ldots \triangleright y_{i_n} : \zeta(\langle [v], [w] \rangle)$ for some index sequence $i_1 \ldots i_n$, then $(v, w) \in \mathsf{PCP}_n$.*

Proof. Induction on n (cf. `pcp_reduction.piped_term_to_pcp_set`).
Basis Step: $\Gamma \vdash_{\mathcal{C}} x : \zeta(\langle [v], [w] \rangle)$ implies $\zeta([v]) = \zeta([w])$. By Lemma 19 we obtain $v = w$.
Inductive Step: Assume $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \ldots \triangleright y_{i_n} \triangleright y_l : \zeta(\langle [v], [w] \rangle)$ for some index sequence $i_1, \ldots, i_n, l$. We necessarily have $\Gamma \vdash_{\mathcal{C}} y_l : \sigma \to \zeta(\langle [v], [w] \rangle)$ and $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \ldots \triangleright y_{i_n} : \sigma$ for some type σ. Additionally, $\sigma \to \zeta(\langle [v], [w] \rangle) = \xi(\langle a \cdot v_l, b \cdot w_l \rangle \to \langle a, b \rangle)$ for some substitution ξ, which implies $\zeta([v]) = \xi(a)$, $\zeta([w]) = \xi(b)$ and $\zeta(\bullet) = \xi(\bullet)$. Therefore, $\xi(a \cdot v_l) = \zeta([vv_l])$ and $\xi(b \cdot w_l) = \zeta([ww_l])$. As a result, we have $\sigma = \xi(\langle a \cdot v_l, b \cdot w_l \rangle) = \zeta(\langle [vv_l], [ww_l] \rangle)$.
By induction hypothesis $(vv_l, ww_l) \in \mathsf{PCP}_n$, which implies $(v, w) \in \mathsf{PCP}_{n+1}$. ◀

Let us define $\mathfrak{n} \in \mathbb{N} \cup \{\infty\}$ (cf. `pcp_reduction.min_special`) as either the minimal size of a combinatory term typable in Γ by $\sigma \to \sigma \to \sigma$ or as ∞ if no such term exists.

$$\mathfrak{n} = \min\{\mathrm{size}(F) \mid \Gamma \vdash_{\mathcal{C}} F : \sigma \to \sigma \to \sigma \text{ for some type } \sigma\}$$

Intuitively, a 'small', i.e. of size less than $\mathfrak{n}$, derivation of an instance of $\langle [v_1], [w_1] \rangle$ contains no derivation of an instance of $\bullet \to \bullet \to \bullet$. Due to our pair encoding, which contains as its first argument the type $\bullet \to \bullet \to \bullet$, we are able to severely restrict the shape of the minimal derivation of $\langle [v_1], [w_1] \rangle$ (cf. Lemma 23).

▶ **Lemma 23.** *If $\Gamma \vdash_{\mathcal{C}} F : \zeta(\langle \sigma, \tau \rangle)$ for some substitution ζ such that $\mathrm{size}(F) < \mathfrak{n}$, then $F = x \triangleright y_{i_1} \triangleright \ldots \triangleright y_{i_m}$ for some (possibly empty) index sequence $i_1, \ldots, i_m$.*

Proof. Induction on $\mathrm{size}(F)$ (cf. `pcp_reduction.small_to_piped_term`).
Basis Step: We have $F \neq z$ and $F \neq y_i$ for any i because the type of the corresponding combinator is not unifiable with $\langle \sigma, \tau \rangle$. If $F = x$ the claim follows.
Inductive Step:
Case $F = z\, G_1 \ldots G_m$ for some $m \geq 1$: We have $\Gamma \vdash_{\mathcal{C}} G_1 : \zeta(\langle [v_1], [w_1] \rangle)$ for some substitution ζ which using $a \mapsto \zeta(\bullet)$ implies $\Gamma \vdash_{\mathcal{C}} z\, G_1 : \zeta(\bullet) \to \zeta(\bullet) \to \zeta(\bullet)$. Therefore, $\mathfrak{n} \leq \mathrm{size}(z\, G_1) \leq \mathrm{size}(F) < \mathfrak{n}$. ↯
Case $F = x\, G_1 \ldots G_m$: We have $\Gamma \vdash_{\mathcal{C}} G_1 : \sigma' \to \sigma' \to \sigma'$ for some σ'. However, $\mathfrak{n} \leq \mathrm{size}(G_1) < \mathrm{size}(F) < \mathfrak{n}$ is a contradiction. ↯
Case $F = y_i\, G$ for some i: We have $\Gamma \vdash_{\mathcal{C}} G : \zeta(\langle \sigma \cdot v_i, \tau \cdot w_i \rangle)$. By induction hypothesis we have $G = x \triangleright y_{i_1} \triangleright \ldots \triangleright y_{i_m}$ for some (potentially empty) index sequence $i_1, \ldots, i_m$. Therefore, $F = x \triangleright y_{i_1} \triangleright \ldots \triangleright y_{i_m} \triangleright y_i$.
Case $F = y_i\, G_1 \ldots G_m$ for some i and some $m \geq 2$: We have $\Gamma \vdash_{\mathcal{C}} G_2 : \sigma' \to \sigma' \to \sigma'$ for some σ'. However, $\mathfrak{n} \leq \mathrm{size}(G_2) < \mathrm{size}(F) < \mathfrak{n}$ is a contradiction. ↯ ◀

Next, we show that if $a \to b \to a$ is derivable, then there is a small derivation of an instance of $\langle [v_1], w_1 \rangle$ (cf. Lemma 24).

▶ **Lemma 24.** *If $|\Gamma| \vdash_{\mathcal{H}} a \to b \to a$, then $\Gamma \vdash_{\mathcal{C}} F : \zeta(\langle [v_1], [w_1] \rangle)$ for some substitution ζ and combinatory term F such that $\mathrm{size}(F) < \mathfrak{n}$.*

Proof. Case analysis (cf. `pcp_reduction.aba_to_small_v1w1`). $|\Gamma| \vdash_{\mathcal{H}} a \to b \to a$ implies $\mathfrak{n} < \infty$, i.e. $\Gamma \vdash_{\mathcal{C}} H : \sigma \to \sigma \to \sigma$ for some type σ and combinatory term H such that $\mathrm{size}(H) = \mathfrak{n}$.

Cases $H = x$ or $H = z$ or $H = y_i$ or $H = y_i\, G$: H cannot be typed by $\sigma \to \sigma \to \sigma$. ↯
Case $H = x\, G_1 \ldots G_m$ for some $m \geq 1$: We have $\Gamma \vdash_{\mathcal{C}} G_1 : \sigma' \to \sigma' \to \sigma'$ for some σ'.
However, $\mathfrak{n} \leq \mathrm{size}(G_1) < \mathrm{size}(H) = n$ is a contradiction. ↯

Case $H = y_i\, G_1 \dots G_m$ for some $m \geq 2$: We have $\Gamma \vdash_{\mathcal{C}} G_2 : \sigma' \to \sigma' \to \sigma'$ for some σ'. Again, $\mathfrak{n} \leq \text{size}(G_2) < \text{size}(H) = n$ is a contradiction. ↯

Case $H = z\, G_1 \dots G_m$ for some $m \geq 1$: We have $\Gamma \vdash_{\mathcal{C}} G_1 : \zeta(\langle [v_1], [w_1] \rangle)$ for some substitution ζ which proves the claim since $\text{size}(G_1) < \text{size}(H) = \mathfrak{n}$. ◀

By construction, elements $(v, w) \in \mathsf{PCP}_n$ are associated with terms of type $\langle [v], [w] \rangle$ (cf. Lemma 25).

▶ **Lemma 25.** *Let $v, w \in \{\mathtt{a}, \mathtt{b}\}^*$. If $(v, w) \in \mathsf{PCP}_n$, then $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \dots \triangleright y_{i_n} : \langle [v], [w] \rangle$ for some index sequence $i_1 \dots i_n$.*

Proof. Routine induction on n (cf. `pcp_reduction.pcp_set_to_piped_term`).

Basis Step: $(v, w) \in \mathsf{PCP}_0$ implies $v = w$. Using the substitution $a \mapsto [v] = [w]$ we obtain $\Gamma \vdash_{\mathcal{C}} x : \langle [v], [w] \rangle$.

Inductive Step: $(v, w) \in \mathsf{PCP}_{n+1}$ implies $(vv_l, ww_l) \in \mathsf{PCP}_n$ for some $l \in \{1, \dots k\}$. By induction hypothesis $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \dots \triangleright y_{i_n} : \langle [vv_l], [ww_l] \rangle$ for some index sequence $i_1 \dots i_n$. Using the substitution $a \mapsto [v], b \mapsto [w]$ we have $\Gamma \vdash_{\mathcal{C}} y_l : \langle [vv_l], [ww_l] \rangle \to \langle [v], [w] \rangle$. By ($\to$E), we obtain the claim $\Gamma \vdash_{\mathcal{C}} x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \dots \triangleright y_{i_n} \triangleright y_l : \langle [v], [w] \rangle$. ◀

Finally, we can prove the following key Lemma 26 (cf. `pcp_reduction.aba_iff_pcp_set`) which relates membership of (v_1, w_1) in some PCP_n and derivability of $a \to b \to a$ from $|\Gamma|$.

▶ **Lemma 26.** *We have $|\Gamma| \vdash_{\mathcal{H}} a \to b \to a$ iff $(v_1, w_1) \in \mathsf{PCP}_n$ for some $n \geq 0$.*

Proof. $\Longrightarrow$: Assume $|\Gamma| \vdash_{\mathcal{H}} a \to b \to a$. By Lemma 24 we have $\Gamma \vdash_{\mathcal{C}} F : \zeta(\langle [v_1], [w_1] \rangle)$ for some substitution ζ and combinatory term F with $\text{size}(F) < \mathfrak{n}$. By Lemma 23 we have $F = x \triangleright y_{i_1} \triangleright y_{i_2} \triangleright \dots \triangleright y_{i_n}$ for some index sequence $i_1 \dots i_n$. Finally, by Lemma 22 we have $(v_1, w_1) \in \mathsf{PCP}_n$.

$\Longleftarrow$: Assume $(v_1, w_1) \in \mathsf{PCP}_n$. By Lemma 25 we have $\Gamma \vdash_{\mathcal{C}} F : \langle [v_1], [w_1] \rangle$ for some term F. Using an appropriate substitution, we obtain $\Gamma \vdash_{\mathcal{C}} z\, F : a \to b \to a$. ◀

Principality of Axioms $|\Gamma|$

In this subsection we inspect evidence that axioms $|\Gamma|$ are, in fact, principal by means of examples. Moreover, we explore the correspondence between compositions of principal inhabitants and solutions to the given PCP instance. Particularly, a composition of principal inhabitants which shows at type level that the given PCP instance has a solution also constructs and verifies the corresponding solution at term level.

Let us fix the following PCP instance

k	$(v_1,$	$w_1)$	$(v_2,$	$w_2)$	$(v_3,$	$w_3)$	v_1	v_2	v_1	v_3	$=$	w_1	w_2	w_1	w_3
3	(ba,	b)	(ab,	aa)	(a,	baa)	ba	ab	ba	a	=	b	aa	b	baa

In this case, $|\Gamma|$ consists of the following five axioms

$$\begin{aligned}
\sigma^x &= \langle a, a\rangle \\
&= (\bullet \to \bullet \to \bullet) \to (a \to a \to \bullet) \to (\bullet \to a) \to (\bullet \to a) \to \bullet \to \bullet \to \bullet \\
\sigma^z &= \langle [v_1], [w_1]\rangle \to \bullet \to a \to \bullet \\
&= \langle (\bullet \to \bullet) \to (\bullet \to \bullet \to \bullet) \to \bullet, (\bullet \to \bullet \to \bullet) \to \bullet\rangle \to \bullet \to a \to \bullet \\
\sigma^y_1 &= \langle a \cdot v_1, b \cdot w_1\rangle \to \langle a, b\rangle \\
&= \langle (\bullet \to \bullet) \to (\bullet \to \bullet \to \bullet) \to a, (\bullet \to \bullet \to \bullet) \to b\rangle \to \langle a, b\rangle \\
\sigma^y_2 &= \langle a \cdot v_2, b \cdot w_2\rangle \to \langle a, b\rangle \\
&= \langle (\bullet \to \bullet \to \bullet) \to (\bullet \to \bullet) \to a, (\bullet \to \bullet) \to (\bullet \to \bullet) \to b\rangle \to \langle a, b\rangle \\
\sigma^y_3 &= \langle a \cdot v_3, b \cdot w_3\rangle \to \langle a, b\rangle \\
&= \langle (\bullet \to \bullet) \to a, (\bullet \to \bullet) \to (\bullet \to \bullet) \to (\bullet \to \bullet \to \bullet) \to b\rangle \to \langle a, b\rangle
\end{aligned}$$

We utilize Haskell[3] to implement principal inhabitants of the above axioms. Further, we use GHC's built-in type inference to attest principality[4]. In the following code fragments '`\f -> M`' represents the λ-term '$\lambda f.M$' and '`--comment`' marks a comment.

The following λ-term pair_{eq} is a principal inhabitant of σ^x and is implemented in the subsequent Listing 1 together with its inferred principal type.

$$\begin{aligned}\text{pair}_{\text{eq}} =& \lambda f.\lambda \text{is}_{vw}.\lambda \text{make}_v.\lambda \text{make}_w.\lambda x.\lambda y. \\ & f\ x\ (f\ (\text{is}_{vw}\ (\text{make}_v\ x)\ (\text{make}_w\ x))\ (\text{is}_{vw}\ (\text{make}_w\ y)\ (\text{make}_v\ y)))\end{aligned}$$

Listing 1 Principal Inhabitant of σ^x.

```
pair_eq = \f -> \is_vw -> \make_v -> \make_w -> \x -> \y ->
    f x (f (is_vw (make_v x) (make_w x)) (is_vw (make_w y) (make_v y)))

--inferred principal type
pair_eq
  :: (t1 -> t1 -> t1)
     -> (t2 -> t2 -> t1) -> (t1 -> t2) -> (t1 -> t2) -> t1 -> t1 -> t1
```

Intuitively, f is used as conjunction and is_{vw}, make_v, make_w are used to attest equality of v and w. Following this intuition, the type $\bullet$ is inhabited by tautologies (e.g. the Haskell value `True`), which leads to a principal inhabitant of σ^z implemented in the following Listing 2.

Listing 2 Principal Inhabitant of σ^z.

```
check_ba_b = \pair -> \x -> \z ->
    let f = pair (\x1 -> \x2 -> x) (\v -> \w -> x)
            (\x' -> \v1c2 -> \v1c1 -> x) (\x' -> \w1c1 -> x)
    in pair
        (\x1 -> \x2 -> f x1 x2) --and
        (\v -> \w -> f (v (f x) f) (w f)) --is_vw
        (\x' -> \v1c2 -> \v1c1 -> f x' (f (v1c2 x) (v1c1 x x))) --make_v
        (\x' -> \w1c1 -> f x' (w1c1 x x)) --make_w
        x --True
        x --True
```

[3] https://www.haskell.org/
[4] Although Haskell's type system is more expressive than simple types, we argue that the additional features play no role in the given examples.

```
check_ba_b --inferred principal type
  :: ((p1 -> p1 -> p1)
      -> (((p1 -> p1) -> (p1 -> p1 -> p1) -> p1)
          -> ((p1 -> p1 -> p1) -> p1) -> p1)
      -> (p1 -> (p1 -> p1) -> (p1 -> p1 -> p1) -> p1)
      -> (p1 -> (p1 -> p1 -> p1) -> p1)
      -> p1 -> p1 -> p1) -> p1 -> p2 -> p1
```

Note that the `let ... in` construction in this case does not contribute to principality (replacing all occurrences of `f` by copies of its implementation does not change the principal type) and is just syntactic sugar. In the above, `f` is used not only as conjunction but also to construct inhabitants `f x` of type $\bullet \to \bullet$ (represents `a`) and `f` of type $\bullet \to \bullet \to \bullet$ (represents `b`). As a result, `v (f x) f` is of type $\bullet$ iff `v` is of type [ba]. Since `f` allows for arbitrary nesting, we argue that the above principal inhabitant construction can be generalized for any pair of words v_1, w_1.

Finally, we implement principal inhabitants of $\sigma_1^y, \sigma_2^y, \sigma_3^y$ in the following Listing 3.

Listing 3 Principal Inhabitants of $\sigma_1^y, \sigma_2^y, \sigma_3^y$.

```
step_ba_b = \pair -> \f -> \is_vw -> \make_v -> \make_w -> \x -> \y ->
    f (f x y) (f (is_vw (make_v x) (make_w y))
        (pair (\x1 -> \x2 -> f x1 x2)
            (\v' -> \w' -> is_vw (v' (f x) f) (w' f)) --is_v'w'
            (\x' -> \v'c2 -> \v'c1 ->
                make_v (f x' (f (v'c2 x) (v'c1 x x)))) --v'
            (\x' -> \w'c1 ->
                make_w (f x' (w'c1 x x))) --w'
            x y))

step_ba_b --inferred principal type
  :: ((t1 -> t1 -> t1)
      -> (((t1 -> t1) -> (t1 -> t1 -> t1) -> t2)
          -> ((t1 -> t1 -> t1) -> t3) -> t1)
      -> (t1 -> (t1 -> t1) -> (t1 -> t1 -> t1) -> t2)
      -> (t1 -> (t1 -> t1 -> t1) -> t3)
      -> t1 -> t1 -> t1)
     -> (t1 -> t1 -> t1) -> (t2 -> t3 -> t1) -> (t1 -> t2) -> (t1 -> t3)
     -> t1 -> t1 -> t1

step_ab_aa = \pair -> \f -> \is_vw -> \make_v -> \make_w -> \x -> \y ->
    f (f x y) (f (is_vw (make_v x) (make_w y))
        (pair (\x1 -> \x2 -> f x1 x2)
            (\v' -> \w' -> is_vw (v' f (f x)) (w' (f x) (f x)))
            (\x' -> \v'c2 -> \v'c1 ->
                make_v (f x' (f (v'c2 x x) (v'c1 x))))
            (\x' -> \w'c2 -> \w'c1 ->
                make_w (f x' (f (w'c2 x) (w'c1 x))))
            x y))

step_a_baa = \pair -> \f -> \is_vw -> \make_v -> \make_w -> \x -> \y ->
    f (f x y) (f (is_vw (make_v x) (make_w y))
        (pair (\x1 -> \x2 -> f x1 x2)
            (\v' -> \w' -> is_vw (v' (f x)) (w' (f x) (f x) f))
            (\x' -> \v'c1 ->
                make_v (f x' (v'c1 x)))
            (\x' -> \w'c3 -> \w'c2 -> \w'c1 ->
                make_w (f x' (f (w'c3 x) (f (w'c2 x) (w'c1 x x)))))
            x y))
```

Similarly, to the principal inhabitant of σ^z, we need to ensure that the principal type of the first argument `pair` is suffixed by the given words. Now, we may (and have to) use the additional arguments `is_vw` of type $a \to b \to \bullet$ and `make_v`, `make_w` of types $\bullet \to a$ and $\bullet \to b$.

Using `f` as conjunction and a way to construct inhabitants of character representations, `\v' -> \w' -> is_vw (v' (f x) f) (w' f)` implements the principal inhabitant of $(a \cdot \mathtt{ba}) \to (b \cdot \mathtt{b}) \to \bullet$. Having `x` of type $\bullet$, `\x' -> \v'c2 -> \v'c1 -> make_v (f x' (f (v'c2 x) (v'c1 x x)))` implements the principal inhabitant of $\bullet \to (a \cdot \mathtt{ba})$. Again, we argue that the principal inhabitant construction can be generalized inductively.

Knowing a solution to the above PCP instance, let us compose the given implementations in the following Listing 4 to an implementation of $a \to b \to a$.

Listing 4 Inhabitant of $a \to b \to a$.

```
(|>) x y = y x
k = pair_eq |> step_a_baa |> step_ba_b |> step_ab_aa |> check_ba_b
k :: p1 -> p2 -> p1 --inferred principal type
```

As expected, evaluating '`k "Lazy␣Hello" undefined`' via GHC results in `"Lazy␣Hello"`, which shows that the second argument `undefined` is not evaluated. For some higher-order pleasure, the expression '`k ((.)$(.)) "Ask␣the␣owl" (==) 2 (1+) 1`' evaluates (as expected) to `True`.

Interestingly, the above principal implementation of the given axioms has more computational meaning than just preserving an argument. Viewing `True` as the only inhabitant of $\bullet$, consider the following piece of code.

Listing 5 Composition of Inhabitants.

```
a = \x -> x
b = \x -> \y -> x && y

--c == a
is_a = \c -> (c True == a True) && (c False == a False)
--c == b
is_b = \c -> (c True True == b True True)
        && (c True False == b True False)
        && (c False True == b False True)
        && (c False False == b False False)

make_ba = \x -> \c2 -> \c1 -> x && (is_a c2) && (is_b c1)
make_b = \x -> \c1 -> x && (is_b c1)

is_ba_b = \v -> \w -> (v a b) && (w b)

composition = pair_eq |> step_a_baa |> step_ba_b |> step_ab_aa
f = composition (&&) is_ba_b make_ba make_b

show_function_table = do
    putStrLn ("f␣True␣␣True␣␣=␣" ++ (show (f True True)))
    putStrLn ("f␣True␣␣False␣=␣" ++ (show (f True False)))
    putStrLn ("f␣False␣True␣␣=␣" ++ (show (f False True)))
    putStrLn ("f␣False␣False␣=␣" ++ (show (f False False)))
```

The terms `a` and `b` implement representations of characters a and b. Given a character `c`, the function `is_a` (resp. `is_b`) extensionally verifies equality to `a` (resp. `b`). The functions `make_ba`, `make_b` and `is_ba_a` construct and verify inhabitants of representations of the corresponding words.

Using the solution to the given PCP instance we have that `composition` is of type

$$\langle [ba], [b] \rangle = (\bullet \to \bullet \to \bullet) \to ([ba] \to [b] \to \bullet) \to (\bullet \to [ba]) \to (\bullet \to [b]) \to \bullet \to \bullet \to \bullet$$

and, therefore, `f` is of type $\bullet \to \bullet \to \bullet$. Displaying the function table of `f` reveals that if conjunction (`&&`) is the first argument of `composition`, then `f` remains a conjunction. On closer examination of the underlying implementation, the term `f True True` evaluates to `True` by constructing the representation of the actual solution to the given PCP instance and verifying the correctness of the constructed solution via `pair_eq`. Viewing $|\Gamma|$ as an intuitionistic axiomatization of the corresponding PCP instance, this provides an additional twist to the Curry-Howard isomorphism. The constructed λ-term not only corresponds to the proof that a formula is derivable, but actually solves the underlying problem on the term level.

Derivability from $a \to b \to a$

We conclude this section by the systematic observation that the condition $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma_i$ for $1 = 1 \dots n$ is decidable. The key observation is that any formula derivable from $a \to b \to a$ is a prefixed instance of $a \to b \to a$ (cf. Lemma 27).

▶ **Lemma 27.**
$[\{a \to b \to a\}]_{\mathcal{H}} = \{\sigma_1 \to \dots \to \sigma_n \to \sigma_{n+1} \to \tau \to \sigma_{n+1} \mid n \geq 0$ *and* $\sigma_1, \dots, \sigma_{n+1}, \tau \in \mathbb{T}\}$

Proof. **$\supseteq$:** n-fold use of Lemma 20 starting with $\{a \to b \to a\} \vdash_{\mathcal{H}} \sigma_{n+1} \to \tau \to \sigma_{n+1}$.
$\subseteq$: Let $\Gamma = \{x : a \to b \to a\}$. Assume $\Gamma \vdash_{\mathcal{C}} F : \sigma$ such that size(F) is minimal. We show the claim by induction on size(F).
Case $F = x$: We have $\sigma = \zeta(a \to b \to a)$ for some substitution ζ, showing the claim.
Case $F = x\ G$: We have $\sigma = \zeta(b \to a)$ and $\Gamma \vdash_{\mathcal{C}} G : \zeta(a)$ for some substitution ζ. By the induction hypothesis $\zeta(a) = \sigma_1 \to \dots \to \sigma_n \to \sigma_{n+1} \to \tau \to \sigma_{n+1}$ for some $n \geq 0$ and $\sigma_1, \dots, \sigma_{n+1}, \tau \in \mathbb{T}$. Therefore, $\sigma = \zeta(b \to a) = \zeta(b) \to \sigma_1 \to \dots \to \sigma_n \to \sigma_{n+1} \to \tau \to \sigma_{n+1}$, which shows the claim.
Case $F = x\ G_1 \dots G_l$ for some $l \geq 2$: We have $\Gamma \vdash_{\mathcal{C}} x : \tau_1 \to \dots \to \tau_l \to \sigma = \zeta(a \to b \to a)$ for some substitution ζ, therefore $\tau_1 = \tau_3 \to \dots \to \tau_l \to \sigma$. Additionally, we have $\Gamma \vdash_{\mathcal{C}} G_i : \tau_i$ for $i = 1 \dots l$. As a result, $\Gamma \vdash_{\mathcal{C}} G_1\ G_3 \dots G_l : \sigma$ with size$(G_1\ G_3 \dots G_l) <$ size(F) which contradicts minimality of size(F). ◀

As a result of the above syntactic characterization by Lemma 27 of formulae derivable from $a \to b \to a$, it is decidable in linear time whether for a given formula σ we have $\sigma \in [\{a \to b \to a\}]_{\mathcal{H}}$. This contrasts PSPACE-completeness to decide $\sigma \in [\{a \to b \to a, (a \to b \to c) \to (a \to b) \to (a \to c)\}]_{\mathcal{H}}$ [18].

4 Recognizing Axiomatizations of $a \to a$

In this section, we record that axiomatizations of the Hilbert-style calculus containing only the axiom $a \to a$ are recognizable in linear time. The key observation is that you cannot meaningfully compose axioms that are instances of $a \to a$. Therefore, the only derivable formulae are instances of the given axioms (cf. Lemma 28).

▶ **Lemma 28.** *Given* $\sigma_1, \dots, \sigma_n \in \mathbb{T}$ *we have*

$$[\{\sigma_1 \to \sigma_1, \dots, \sigma_n \to \sigma_n\}]_{\mathcal{H}} = \bigcup_{i=1}^{n} \{\zeta(\sigma_i \to \sigma_i) \mid \zeta \textit{ is a substitution}\}$$

Proof. **$\supseteq$:** holds by instantiation of $\sigma_i \to \sigma_i$ for $i = 1 \dots n$.
$\subseteq$: Let $\Gamma = \{x_1 : \sigma_1 \to \sigma_1, \dots, x_n : \sigma_n \to \sigma_n\}$. Assume $\Gamma \vdash_{\mathcal{C}} F : \sigma$ such that size(F) is minimal.
Case $F = x_i$ for some $i \in \{1, \dots, n\}$: We have $\sigma = \zeta(\sigma_i \to \sigma_i)$ for some substitution ζ, which shows the claim.

Case $F = x_i\ G_1 \dots G_l$ for some $i \in \{1, \dots, n\}$: We have $\Gamma \vdash_{\mathcal{C}} x_i : \tau_1 \to \dots \to \tau_l \to \sigma = \zeta(\sigma_i \to \sigma_i)$ for some substitution ζ, therefore $\tau_1 = \tau_2 \to \dots \to \tau_l \to \sigma$. Additionally, we have $\Gamma \vdash_{\mathcal{C}} G_i : \tau_i$ for $i = 1 \dots l$. As a result, $\Gamma \vdash_{\mathcal{C}} G_1\ G_2 \dots G_l : \sigma$ with $\mathrm{size}(G_1\ G_2 \dots G_l) < \mathrm{size}(F)$ which contradicts minimality of $\mathrm{size}(F)$. ◀

As a side note, the above proof of Lemma 28 implies that the set of minimal proofs from axioms $\{\sigma_1 \to \sigma_1, \dots, \sigma_n \to \sigma_n\}$ is of finite cardinality n.

▶ **Corollary 29.** $[\{a \to a\}]_{\mathcal{H}} = \{\tau \to \tau \mid \tau \in \mathbb{T}\}$.

▶ **Lemma 30.** *If $[\Delta]_{\mathcal{H}} = [\{a \to a\}]_{\mathcal{H}}$, then $b \to b \in \Delta$ for some $b \in \mathbb{A}$.*

Proof. Since $[\{a \to a\}]_{\mathcal{H}} \supseteq [\Delta]_{\mathcal{H}}$ implies $[\{a \to a\}]_{\mathcal{H}} \supseteq \Delta$ we have $\Delta = \{\sigma_1 \to \sigma_1, \dots, \sigma_n \to \sigma_n\}$ for some $\sigma_1, \dots, \sigma_n \in \mathbb{T}$ by Corollary 29. By Lemma 28 we obtain $[\Delta]_{\mathcal{H}} = \bigcup_{i=1}^{n} \{\zeta(\sigma_i \to \sigma_i) \mid \zeta \text{ is a substitution}\}$. Due to $[\{a \to a\}]_{\mathcal{H}} \subseteq [\Delta]_{\mathcal{H}}$ we obtain $a \to a = \zeta(\sigma_i \to \sigma_i)$ for some $i \in \{1, \dots, n\}$ and some substitution ζ, which holds iff $\sigma_i \to \sigma_i = b \to b$ for some $b \in \mathbb{A}$. ◀

▶ **Corollary 31.** *We have $[\Delta]_{\mathcal{H}} = [\{a \to a\}]_{\mathcal{H}}$ iff $\Delta \subseteq \{\sigma \to \sigma \mid \sigma \in \mathbb{T}\}$ and $b \to b \in \Delta$ for some $b \in \mathbb{A}$.*

As a result of Corollary 31, recognizing axiomatizations of $a \to a$ is decidable in linear time.

5 Recognizing Axiomatizations of $a \to b \to b$

In this section, we extend linear time recognizability to axiomatizations of the Hilbert-style calculus containing only the axiom $a \to b \to b$. Similarly to $a \to a$, meaningful logical compositions of instances of $a \to b \to b$ are limited (cf. Lemma 32).

▶ **Lemma 32.** $[\{a \to b \to b\}]_{\mathcal{H}} = \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}\}$

Proof. **$\supseteq$:** Instantiation resp. derivability of $a \to a$.
$\subseteq$: Let $\Gamma = \{x : a \to b \to b\}$. Assume $\Gamma \vdash_{\mathcal{C}} F : \sigma$ such that $\mathrm{size}(F)$ is minimal.

Case $F = x$: We have $\sigma = \zeta(a \to b \to b)$ for some substitution ζ, which shows the claim.

Case $F = x\ G$: We have $\Gamma \vdash_{\mathcal{C}} x : \tau \to \sigma = \zeta(a \to b \to b)$ for some substitution ζ, therefore $\sigma = \sigma' \to \sigma'$, which shows the claim.

Case $F = x\ G_1 \dots G_l$ where $l \geq 2$: We have $\Gamma \vdash_{\mathcal{C}} x : \tau_1 \to \dots \to \tau_l \to \sigma = \zeta(a \to b \to b)$ for some substitution ζ, therefore $\tau_3 = \tau_3 \to \dots \to \tau_l \to \sigma$. Additionally, we have $\Gamma \vdash_{\mathcal{C}} G_i : \tau_i$ for $i = 1 \dots l$. As a result, $\Gamma \vdash_{\mathcal{C}} G_2\ G_3 \dots G_l : \sigma$ with $\mathrm{size}(G_2\ G_3 \dots G_l) < \mathrm{size}(F)$ which contradicts minimality of $\mathrm{size}(F)$. ◀

As a side note, the above proof of Lemma 32 implies that the set of minimal proofs starting from the axiom $a \to b \to b$ is finite (in fact of cardinality 2).

Using the above observation, we can characterize axiomatizations of $a \to b \to b$ syntactically (cf. Lemma 33).

▶ **Lemma 33.** *We have $[\Delta]_{\mathcal{H}} = [\{a \to b \to b\}]_{\mathcal{H}}$ iff $c \to d \to d \in \Delta$ for some $c, d \in \mathbb{A}$ and $\Delta \subseteq \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}\}$.*

Proof. $\Longleftarrow$: $\sigma \to \tau \to \tau$ and $\tau \to \tau$ are derivable from $c \to d \to d$ for any $\sigma, \tau \in \mathbb{T}$.

$\Longrightarrow$: Due to $[\{a \to b \to b\}]_{\mathcal{H}} \supseteq \Delta$ we have $\Delta \subseteq \{\sigma \to \tau \to \tau \mid \sigma, \tau \in \mathbb{T}\} \cup \{\tau \to \tau \mid \tau \in \mathbb{T}\}$ by Lemma 32. Due to $[\{a \to b \to b\}]_{\mathcal{H}} \subseteq [\Delta]_{\mathcal{H}}$ we obtain $\Delta \vdash_{\mathcal{H}} a \to b \to b$. By case analysis (similar to the proof of Lemma 32) the minimal derivation of $\Delta \vdash_{\mathcal{H}} a \to b \to b$ is an instantiation of some $\sigma \to \tau \to \tau \in \Delta$, i.e. $a \to b \to b = \zeta(\sigma \to \tau \to \tau)$ for some substitution ζ. Therefore, $\sigma \to \tau \to \tau = c \to d \to d$ for some $c, d \in \mathbb{A}$. ◀

As a result of the above Lemma 33, recognizing axiomatizations of $a \to b \to b$ is decidable in linear time. One reason why recognizing axiomatizations of $a \to a$ and $a \to b \to b$ is trivial is that the set of minimal proofs in the corresponding calculi is finite, which is not the case for $a \to b \to a$.

6 Conclusion

We have shown that even under two severe restrictions subintuitionistic propositional calculi have undecidable derivability. In particular, it is undecidable whether from a given set of axioms (all of which are principal and derivable from $a \to b \to a$) the formula $a \to b \to a$ is derivable. In contrast, with respect to the formula $a \to b \to b$ derivability is decidable in linear time. Our result sheds some light on the lower end of the spectrum of propositional calculi. In future, it may be of systematic interest to inspect (sets of) axioms that correspond to principal types of other well known combinators such as **S**.

Under the Curry-Howard isomorphism, our result is related to type-based composition synthesis. Particularly, even under the the assumption that the given building blocks are 'natural' (in the sense of principality) and 'plain' (in the sense of their types are derivable from the axiom $a \to b \to a$) synthesis remains undecidable. The research program in type-based composition synthesis outlined in [15] is based on bounded variants of the inhabitation problem in combinatory logic [8].

Additionally, the reduction from the Post correspondence problem proving of our main result is formalized in the Lean theorem prover.

References

1 H. P. Barendregt. *The Lambda Calculus. Its Syntax and Semantics.* Studies in Logic and the Foundations of Mathematics, 2nd Edition. Elsevier Science Publishers, 1984. `doi:10.2307/2274112`.

2 H. P. Barendregt, W. Dekkers, and R. Statman. *Lambda Calculus with Types.* Perspectives in Logic, Cambridge University Press, 2013.

3 Marcin Benke, Aleksy Schubert, and Daria Walukiewicz-Chrzaszcz. Synthesis of Functional Programs with Help of First-Order Intuitionistic Logic. In *FSCD 2016*, volume 52 of *LIPIcs*, pages 12:1–12:16, 2016. `doi:10.4230/LIPIcs.FSCD.2016.12`.

4 Grigoriy V. Bokov. Completeness problem in the propositional calculus. *Intelligent Systems*, 13(1-4):165–182, 2009.

5 Grigoriy V. Bokov. Undecidability of the problem of recognizing axiomatizations for propositional calculi with implication. *Logic Journal of the IGPL*, 23(2):341–353, 2015. `doi:10.1093/jigpal/jzu047`.

6 Grigoriy V. Bokov. Undecidable problems for propositional calculi with implication. *Logic Journal of the IGPL*, 24(5):792–806, 2016. `doi:10.1093/jigpal/jzw013`.

7 Sabine Broda and Luís Damas. On Long Normal Inhabitants of a Type. *J. Log. Comput.*, 15(3):353–390, 2005. `doi:10.1093/logcom/exi016`.

8 Boris Düdder, Moritz Martens, Jakob Rehof, and Paweł Urzyczyn. Bounded Combinatory Logic. In *CSL 2012, Proceedings of Computer Science Logic*, volume 16 of *LIPIcs*, pages 243–258. Schloss Dagstuhl, 2012.

9 Andrej Dudenhefner and Jakob Rehof. The Complexity of Principal Inhabitation. In *FSCD 2017*, pages 15:1–15:14, 2017. `doi:10.4230/LIPIcs.FSCD.2017.15`.

10 J. R. Hindley. *Basic Simple Type Theory.* Cambridge Tracts in Theoretical Computer Science, vol. 42, Cambridge University Press, 2008.

11 J. R. Hindley and J. P. Seldin. *Lambda-calculus and Combinators, an Introduction.* Cambridge University Press, 2008.

12 AV Kuznecov and E Mendelson. Undecidability of the general problems of completeness, decidability and equivalence for propositional calculi. *The Journal of Symbolic Logic*, 1972.

13 Samuel Linial and Emil L. Post. Recursive Unsolvability of the Deducibility, Tarski's Completeness and Independence of Axioms Problems of Propositional Calculus. *Bulletin of the American Mathematical Society*, 55:50, 1949.

14 Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 1946.

15 Jakob Rehof. Towards Combinatory Logic Synthesis. In *BEAT'13, 1st International Workshop on Behavioural Types*. ACM, 2013.

16 W. E. Singletary. Many-one Degrees Associated with Partial Propositional Calculi. *Notre Dame Journal of Formal Logic*, XV(2):335–343, 1974.

17 M.H. Sørensen and P. Urzyczyn. *Lectures on the Curry-Howard Isomorphism*, volume 149 of *Studies in Logic and the Foundations of Mathematics*. Elsevier, 2006.

18 R. Statman. Intuitionistic Propositional Logic Is Polynomial-space Complete. *Theoretical Computer Science*, 9:67–72, 1979.

19 Mary Katherine Yntema. A detailed argument for the Post-Linial theorems. *Notre Dame Journal of Formal Logic*, 5(1):37–50, 1964. `doi:10.1305/ndjfl/1093957737`.

20 Evgeny Zolin. Undecidability of the Problem of Recognizing Axiomatizations of Superintuitionistic Propositional Calculi. *Studia Logica*, 102(5):1021–1039, 2014. `doi:10.1007/s11225-013-9520-5`.

Proof Terms for Generalized Natural Deduction

Herman Geuvers
Radboud University Nijmegen & Technical University Eindhoven, The Netherlands
herman@cs.ru.nl

Tonny Hurkens
Haps, The Netherlands
hurkens@science.ru.nl

Abstract

In previous work it has been shown how to generate natural deduction rules for propositional connectives from truth tables, both for classical and constructive logic. The present paper extends this for the constructive case with proof-terms, thereby extending the Curry-Howard isomorphism to these new connectives. A general notion of conversion of proofs is defined, both as a conversion of derivations and as a reduction of proof-terms. It is shown how the well-known rules for natural deduction (Gentzen, Prawitz) and general elimination rules (Schroeder-Heister, von Plato, and others), and their proof conversions can be found as instances. As an illustration of the power of the method, we give constructive rules for the `nand` logical operator (also called *Sheffer stroke*).

As usual, conversions come in two flavours: either a *detour conversion* arising from a *detour convertibility*, where an introduction rule is immediately followed by an elimination rule, or a *permutation conversion* arising from an *permutation convertibility*, an elimination rule nested inside another elimination rule. In this paper, both are defined for the general setting, as conversions of derivations and as reductions of proof-terms. The properties of these are studied as proof-term reductions. As one of the main contributions it is proved that detour conversion is strongly normalizing and permutation conversion is strongly normalizing: no matter how one reduces, the process eventually terminates. Furthermore, the combination of the two conversions is shown to be weakly normalizing: one can always reduce away all convertibilities.

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

Keywords and phrases constructive logic, natural deduction, detour conversion, permutation conversion, normalization Curry-Howard isomorphism

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.3

Acknowledgements We thank Iris van der Giessen and the anonymous referees for their valuable comments on the earlier version of this paper.

1 Introduction

Natural deduction rules come in various forms, where the tree format is the most well-known. One either puts formulas A as the nodes and leaves of the tree, or sequents $\Gamma \vdash A$, where Γ is a sequence or a finite set of formulas. Other formalisms use a linear format, using flags or boxes to explicitly manage the open and discharged assumptions.

We [7] use a natural deduction in sequent calculus style, where in addition all rules have a special form:

$$\frac{\dots \quad \Gamma \vdash A_i \quad \dots \qquad \dots \quad \Gamma, A_j \vdash D \quad \dots}{\Gamma \vdash D}$$

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

1. $\Gamma, A_j \vdash D$: we still need to prove D from Γ, but we are now also allowed to use A_j as additional assumption. We call A_j a case.
2. $\Gamma \vdash A_i$: in stead of proving D from Γ, we now need to prove A_i from Γ. We call A_i a lemma.

Given the restricted format of the rules, we don't have to give Γ explicitly, as it can be retrieved from the other information in a deduction. So, the deduction rules are presented without Γ, in the following format

$$\frac{\ldots \quad \vdash A_i \quad \ldots \qquad \ldots \quad A_j \vdash D \quad \ldots}{\vdash D}$$

In [7] we have shown how to derive natural deduction rules for a connective form its definition by a truth table, both for the classical and the intuitionistic case. In that paper, we have shown that the intuitionistic rules are indeed constructive by providing a Kripke semantics. In the present paper we provide a proof-theoretic study of the natural deduction rules for the intuitionistic case. We define a notion of convertibility and conversion for the general connectives, which we analyze by interpreting derivations as proof-terms. So we extend the Curry-Howard isomorphism, that interprets formulas as types and derivations as terms, to include all these new intuitionistic connectives.

It turns out that the standard format for the deduction rules we have chosen (as described above) is very suitable for defining convertibilities and conversion in general, for giving a term interpretation to derivations and for defining reductions on these proof-terms that correspond with conversion (both detour conversion and permutation conversion). The format of our rules also allows the transformation of other formalisms, like the very well-known ones by Gentzen and Prawitz [6, 14] but also more recent ones by Von Plato [23], in terms of ours. This transformation we will define on the proof-term level and we will show how *detour conversion* (the elimination of a *direct convertibility*, an introduction rule immediately followed by an elimination rule) is preserved by the translation.

Standard questions about logic are consistency and decidability. We prove that both hold (in general for our connectives) by proving *weak normalization* for the combined process of *detour conversion* and *permutation conversion.* A permutation conversion operates on a *permutation convertibility*, which arises when an elimination rule blocks a detour convertibility for another connective; in that case one has to permute one elimination rule over another. Weak normalization states that for any derivation (proof-term) we can eliminate convertibilities in such a way that eventually no convertibilities are left. Using this one can prove the sub-formula property and consistency and decidability. We prove weak normalization for the proof-terms by studying reduction of proof-terms.

The interest of our work lies in the fact that the natural deduction rules can be defined and analyzed in such a generic way, capturing very many known instances of deduction rules for intuitionistic logic, but also new deduction rules for new connectives. The key concepts that make this work are our general rule format (described above) and the fact that our system provides natural deduction rules for each connective *in isolation*: rules for one connective do not use another connective. We will illustrate this by giving the nand operator as an extended example. We describe its constructive derivation rules, as they arise from the truth tables. These rules are self-contained, so they only refer to nand itself, and we show how to interpret intuitionistic proposition logic in the logic with only nand. We also give the proof-terms and reductions for nand.

1.1 Related work and contribution of the paper

Natural deduction has been studied extensively, since the original work by Gentzen [6], both for classical and intuitionistic logic. Overviews can be found in [22] and [12]. Also the generalization of natural deduction to include other connectives or allow different derivation rules has been studied by various researchers. Notably, there is the work of Schroeder-Heister [17], Read [16], Tennant [21], Von Plato [23, 12], Milne [11], Francez and Dyckhoff [4, 3] that is related to ours. Schroeder-Heister studies general formats of natural deduction where also rules may be discharged (as opposed to the normal situation where only formulas may be discharged). He also studies a general rule format for intuitionistic logic and shows that the connectives $\wedge, \vee, \to, \bot$ are complete for it. Von Plato, Milne, Francez and Dyckhoff, Read and Tennant study "general elimination rules", where the idea is that elimination rules arise naturally from the introduction rules, following Prawitz's [15] inversion principle: "the conclusion obtained by an elimination does not state anything more than what must have already been obtained if the major premise of the elimination was inferred by an introduction". The elimination rules obtained have the same flavor as the elimination rules we derive from truth tables: the conclusion of elimination Φ is not a sub-formula of Φ, but a general formula D, where there are additional hypothesis that connect Φ and D. For the standard intuitionistic connectives the general elimination rules are quite close to ours, but $\wedge$-elimination is slightly different. Von Plato [23], Lopez-Escobar [10] and Tennant [21] study the standard intuitionistic connectives with general rules.

A difference is that we focus not so much on the rules but on the fact that we can define different and new connectives constructively. In our work, we do not take the introduction rules as primary, with the elimination rules defined from them, but we derive elimination and introduction rules directly from the truth table. Then we optimize them, which can be done in various ways, where we adhere to a fixed format for the rules. Many of the general elimination rules, for example for $\wedge$, appear naturally as a consequence of our approach of deriving the rules from the truth table.

The idea of deriving deduction rules from the truth table also occurs in the work of Milne [11], but in a slightly different way: from the introduction rules, a truth table is derived and then the classical elimination rules are derived from the truth table. For the if-then-else connective, this amounts to classical rules equivalent to ours in [7], but not optimized. We start from the truth table and derive rules for intuitionistic logic.

As remarked, the main contribution of this paper is a proof-theoretic analysis of our system of generalized natural deduction via the Curry-Howard isomorphism that interprets derivations as proof terms and conversions as reductions. We show that many known conversions and reductions are captured by our approach and we prove general normalization results. These is a lot of related work on the Curry-Howard isomorphism that our work rests on, for which we refer to [18, 8].

The present paper builds on research reported in [7]. To make this paper self-contained, we include the definitions and some basic results and examples from [7]: Section 2 repeats the main definitions of [7] in slightly expanded form, where Section 2.1 adds the new example of the nand-connective (Sheffer stroke), which is worked out in detail, especially the connection between nand-logic and intuitionistic proposition logic. Section 3 defines detour conversion and permutation conversion on derivations; the second is new. Section 4 defines the Curry-Howard isomorphism for our general natural deduction format and gives (general) proof terms for natural deductions and reduction rules on them. Section 5 shows how the general rules relate to so called "optimized" rules, which are the ones that are known from the literature for natural deduction and for proof-terms. Section 6 proves normalization results for the calculi of proof-terms. Sections 4, 5, 6 are all new; Section 2.1 is largely new and Section 3 is partially new.

2 Deriving constructive natural deduction rules from truth tables

To make this paper self contained and to fix notions and notations, we recap the main definitions from [7] and explain in detail how the elimination and introduction rules for a connective are derived from its truth table. The elimination rules have the following form. Φ is the formula we eliminate. We have $\Phi = c(A_1, \ldots, A_n)$ where c is a connective of arity n and $n = k + \ell$. The formula D is arbitrary.

$$\frac{\vdash \Phi \quad \vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash D \quad \ldots \quad A_{j_\ell} \vdash D}{\vdash D}\,\text{el}$$

So, $A_{i_1}, \ldots, A_{i_k}, A_{j_1}, \ldots, A_{j_\ell}$ are the direct subformulas of $\Phi = c(A_1, \ldots, A_n)$, where some appear as "lemma" and others as "case" in the derivation rule. The (intuitionistic) introduction rules have the following form. Again, c is a connective of arity n, $\Phi = c(A_1, \ldots, A_n)$ and $n = k + \ell$. (Of course, every rule has its own specific sequence $i_1, \ldots, i_k, j_1, \ldots j_\ell$.)

$$\frac{\vdash A_{i_1} \quad \ldots \quad \vdash A_{i_k} \qquad A_{j_1} \vdash \Phi \quad \ldots \quad A_{j_\ell} \vdash \Phi}{\vdash \Phi}\,\text{in}$$

For a concrete connective c, we derive the elimination and introduction rules from the truth table, as described in the following Definition, taken from [7].

▶ **Definition 1.** Given an n-ary connective c with a truth table t_c (with 2^n rows). We write $\varphi = c(p_1, \ldots, p_n)$, where $p_1, \ldots, p_n$ are proposition letters and we write $\Phi = c(A_1, \ldots, A_n)$, where $A_1, \ldots, A_n$ are arbitrary propositions. Each row of t_c gives rise to an elimination rule or an introduction rule for c in the following way.

$$\begin{array}{ccc|c} p_1 & \ldots & p_n & c(p_1, \ldots, p_n) \\ \hline a_1 & \ldots & a_n & 0 \end{array} \quad \mapsto \quad \frac{\vdash \Phi \quad \ldots \vdash A_j (\text{if } a_j = 1) \ldots \quad \ldots A_i \vdash D (\text{if } a_i = 0) \ldots}{\vdash D}\,\text{el}$$

$$\begin{array}{ccc|c} p_1 & \ldots & p_n & c(p_1, \ldots, p_n) \\ \hline b_1 & \ldots & b_n & 1 \end{array} \quad \mapsto \quad \frac{\ldots \vdash A_j (\text{if } b_j = 1) \ldots \quad \ldots A_i \vdash \Phi (\text{if } b_i = 0) \ldots}{\vdash \Phi}\,\text{in}$$

If $a_j = 1$ in t_c, then A_j occurs as a lemma in the rule; if $a_i = 0$ in t_c, then A_i occurs as a case. The rules are given in abbreviated form and it should be understood that all judgments can be used with an extended hypotheses set Γ. So the elimination rule in full reads as follows (where Γ is a set of propositions).

$$\frac{\Gamma \vdash \Phi \quad \ldots \Gamma \vdash A_j \ (\text{if } a_j = 1) \ldots \quad \ldots \Gamma, A_i \vdash D \ (\text{if } a_i = 0) \ldots}{\Gamma \vdash D}\,\text{el}$$

In an elimination rule, we call $\vdash \Phi$ the *major premise* and the other hypotheses of the rule we call the *minor premises*.

▶ **Definition 2.** Given a set of connectives $\mathcal{C} := \{c_1, \ldots, c_n\}$, we define the *intuitionistic natural deduction system* for $\mathcal{C}$, $\mathsf{IPC}_{\mathcal{C}}$, by the following derivation rules.

- The *axiom rule*

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

- The elimination rules for the connectives in $\mathcal{C}$ and the intuitionistic introduction rules for the connectives in $\mathcal{C}$, as given in Definition 1.

We write $\Gamma \vdash_{\mathcal{C}} A$ if $\Gamma \vdash A$ is derivable using the derivation rules of $\mathsf{IPC}_{\mathcal{C}}$.

▶ **Example 3.**

A	B	$A \vee B$	$A \wedge B$	$A \to B$	$\neg A$
0	0	0	0	1	1
0	1	1	0	1	1
1	0	1	0	0	0
1	1	1	1	1	0

1. From the truth table for $\vee$ we derive the following intuitionistic rules for $\vee$. We label the rules by the relevant entries in the truth table.

$$\frac{\vdash A \vee B \qquad A \vdash D \qquad B \vdash D}{\vdash D}\ \vee\text{-el} \qquad \frac{A \vdash A \vee B \qquad \vdash B}{\vdash A \vee B}\ \vee\text{-in}_{01}$$

$$\frac{\vdash A \qquad B \vdash A \vee B}{\vdash A \vee B}\ \vee\text{-in}_{10} \qquad \frac{\vdash A \qquad \vdash B}{\vdash A \vee B}\ \vee\text{-in}_{11}$$

These rules are all intuitionistically correct, as one can observe by inspection. We will show that these are equivalent to the well-known intuitionistic rules. We will also show how these rules can be optimized and be reduced to 1 elimination rule and 2 introduction rules, which are the well-known ones.

2. From the truth table for $\wedge$ we derive the following intuitionistic rules for $\wedge$, 3 elimination rules and one introduction rule.

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

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

These rules are all intuitionistically correct, as one can observe by inspection. We will show that these are equivalent to the well-known intuitionistic rules. We will also show how these rules can be optimized and be reduced to 2 elimination rules and 1 introduction rule, which are the well-known ones. The elimination rules for $\wedge$ have a bit the flavor of the so called "general elimination rules" of Schroeder-Heister [17] and Von Plato [23], in the sense that we don't derive A, respectively B, from $A \wedge B$, but an auxiliary conclusion D is derived. This rule, also called the *parallel elimination rule* by Tennant [21], is as follows.

$$\frac{\vdash A \wedge B \qquad A, B \vdash D}{\vdash D}\ \wedge\text{-el}^{\text{par}}$$

We will show that this rule can be derived from ours. See Definition 45 and Lemma 46, where this is shown using proof-terms.

3. From the truth table for $\neg$ we also derive the following rules for $\neg$, one elimination rule and one introduction rule.

$$\frac{A \vdash \neg A}{\vdash \neg A}\ \neg\text{-in} \qquad \frac{\vdash \neg A \qquad \vdash A}{\vdash D}\ \neg\text{-el}$$

The elimination rule is familiar. For the introduction rule: to prove $\neg A$, one "only" has to prove $\neg A$ from A, which may seem limited. The traditional $\neg$-in rule is the following.

$$\frac{A \vdash \neg B \qquad A \vdash B}{\vdash \neg A}\ \neg\text{-in}^{\text{t}}$$

The two ¬-introduction rules are equivalent, which we will show in detail (using proof terms) in Lemma 53. To derive ¬-in^{t} from ¬-in one also needs ¬-el, so we view ¬-in as more primitive then the traditional rule ¬-in^{t}.
As an example of the intuitionistic derivation rules for ¬ we show that $A \vdash \neg\neg A$ is derivable:

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

4. From the truth table for → we derive the following intuitionistic rules for →.

$$\dfrac{A \vdash A \to B \qquad B \vdash A \to B}{\vdash A \to B}\,\to\text{-in}_{00} \qquad \dfrac{\vdash A \to B \qquad \vdash A \qquad B \vdash D}{\vdash D}\,\to\text{-el}$$

$$\dfrac{A \vdash A \to B \qquad \vdash B}{\vdash A \to B}\,\to\text{-in}_{10} \qquad \dfrac{\vdash A \qquad \vdash B}{\vdash A \to B}\,\to\text{-in}_{11}$$

These rules are all intuitionistically correct, as one can verify by inspection. For example, for →-in$_{01}$, observe that if $A \vdash A \to B$, then $\vdash A \to B$, so the second hypothesis is superfluous. Similarly for →-in$_{11}$, the first hypothesis is superfluous. We will show that these rules are equivalent to the well-known intuitionistic rules. We will also show how these rules can be optimized and be reduced to 1 elimination rule and 2 introduction rules. These are not the well-known ones, because the well-known →-in-rule does not fit into our scheme:

$$\dfrac{A \vdash B}{\vdash A \to B}\,\to\text{-in}$$

In this rule, both the conclusion is changed *and* an assumption (case) is added. In our system, each rule has the property that a hypothesis either adds an assumption or changes the conclusion (while retaining the same set of assumptions), and this “or” is exclusive.

We continue this section with some more basic properties and notions, most of which have been described briefly in [7]. We also introduce some further notation.

In the logic $\mathsf{IPC}_{\mathcal{C}}$ (Definitions 1 and 2) we can freely reuse formulas and weaken the context, so the structural rules of contraction and weakening are wired into the system. Because weakening is used a lot, we formulate it as a Lemma. The proof is an immediate induction on the derivation.

▶ **Lemma 4** (Weakening). *If $\Gamma \vdash A$ with derivation Π and $\Gamma \subseteq \Delta$, then $\Delta \vdash A$ with derivation Π.*

In natural deduction in tree format, the elimination of a detour convertibility involves *composition* of derivations: the placing of one derivation on top of another, replacing a discharged leaf A on top of a derivation tree (an assumption) by a derivation of A. In natural deduction in sequent calculus style, this amounts to replacing an axiom $\Gamma, A \vdash A$, that appears as the leaf of a derivation tree, by a derivation of $\Delta \vdash A$, where $\Delta \subset \Gamma$. We first define more precisely how the composition of derivation works in natural deduction in sequent calculus style.

▶ **Lemma 5.** *If $\Delta, \varphi \vdash \psi$, and $\Gamma \vdash \varphi$, then $\Gamma, \Delta \vdash \psi$*

Proof. By induction on the derivation of $\Delta, \varphi \vdash \psi$, using weakening (Lemma 4). ◀

To be a bit more precise about what happens with the derivations in the proof of Lemma 5, let Π be the derivation of $\Delta, \varphi \vdash \psi$. Then, due to the format of our rules:

- The only place in Π where the hypothesis φ is actually used is at a leaf of Π, in an instance of the (axiom) rule.
- Contexts can only grow when we walk upwards in a derivation, so these leaves are of the form $\Delta', \varphi \vdash \varphi$ for some $\Delta' \supseteq \Delta$.

We replace this leaf by Σ, the derivation of $\Gamma \vdash \varphi$. Due to weakening, this Σ is also a derivation of $\Gamma, \Delta' \vdash \varphi$, so Π with the leaves of the form $\Delta', \varphi \vdash \varphi$ replaced by Σ yields a correct derivation of $\Gamma, \Delta \vdash \psi$.

▶ **Notation 6.** *If Π is a derivation of $\Delta, \varphi \vdash \psi$ and Σ is a derivation of $\Gamma \vdash \varphi$, then we have a derivation of $\Gamma, \Delta \vdash \psi$ that looks like this:*

$$\begin{array}{c} \begin{array}{ccc} \vdots\,\Sigma & & \vdots\,\Sigma \\ \Gamma \vdash \varphi & \ldots & \Gamma \vdash \varphi \end{array} \\ \vdots\,\Pi \\ \Gamma, \Delta \vdash \psi \end{array}$$

So in Π, every application of an (axiom) rule at a leaf, deriving $\Delta' \vdash \varphi$ for some $\Delta' \supseteq \Delta$ is replaced by a copy of a derivation Σ, which is also a derivation of $\Delta', \Gamma \vdash \varphi$.

The fact that we have weakening supports the following convention.

▶ **Convention 7.** *In examples, to simplify derivations we will often use the following format for an elimination rule (and similarly for an introduction rule).*

$$\frac{\Gamma_0 \vdash \Phi \qquad \ldots \Gamma_j \vdash A_j \ \ (\textit{if } a_j = 1) \ldots \qquad \ldots \Gamma_i, A_i \vdash D \ \ (\textit{if } a_i = 0) \ldots}{\cup_{k=0}^{n} \Gamma_k \vdash D}\ \textit{el}$$

This prevents us from having to copy the full Γ from the conclusion to the hypotheses in a rule; we can limit ourselves to the parts of Γ that we need for that particular branch in the derivation.

We now recall from [7] two lemmas that allow to reduce the number of deduction rules: some rules can be taken together and one or more of the hypotheses can be dropped. For completeness, we give these lemmas again here (Lemma 9 and Lemma 12), with their proofs. First, we motivate Lemma 9 by looking at the example of the rules for $\wedge$ (Example 3).

▶ **Example 8.** From the truth table we have derived the following 3 intuitionistic elimination rules for $\wedge$.

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

$$\frac{\vdash A \wedge B \qquad \vdash A \qquad B \vdash D}{\vdash D}\ \wedge\text{-el}_{10}$$

These rules can be reduced to the following 2 equivalent elimination rules. The index in the rule indicates where it originates from: $\wedge$-el$_{0\_}$ is the combination of $\wedge$-el$_{00}$ and $\wedge$-el$_{01}$.

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

It can be shown that these sets of rules are equivalent. Here we only show the derivability of the ∧-el$_{0\_}$ rule from the rules ∧-el$_{00}$ and ∧-el$_{01}$. As usual, for notational simplicity we suppress the context Γ. Suppose we have derivations of $\vdash A \wedge B$ and of $A \vdash D$. Then we have the following derivation.

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

Note that the third and fourth hypothesis come from the first and second through weakening, and the fifth hypothesis is the axiom rule

The general method here is that we can replace two rules that only differ in one hypothesis, which in one rule occurs as a lemma and in the other as a case, by one rule where the hypothesis is removed. It will be clear that the Γ's above are not relevant for the argument, so we will not write these.

▶ **Lemma 9.** *A system with two derivation rules of the form*

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

is equivalent to the system with these two rules replaced by

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

Proof. The implication from bottom to top is immediate. From top to bottom, suppose we have the two given rules. We now derive the bottom one. Assume we have derivations of $\vdash A_1, \dots, \vdash A_n, B_1 \vdash D, \dots, B_m \vdash D$. We now have the following derivation of $\vdash D$.

$$\dfrac{\vdash A_1 \ \dots\ \vdash A_n \quad B_1 \vdash D \ \dots\ B_m \vdash D \quad \dfrac{A \vdash A_1 \ \dots\ A \vdash A_n \quad A \vdash A \quad A, B_1 \vdash D \ \dots\ A, B_m \vdash D}{A \vdash D}}{\vdash D}$$

◀

Lemma 9 can be applied to elimination and introduction rules. An application to elimination rules is given in Example 8. We now give two applications to introduction rules.

▶ **Example 10.** From the truth table we have derived the following 3 intuitionistic introduction rules for ∨.

$$\dfrac{A \vdash A \vee B \qquad \vdash B}{\vdash A \vee B}\ \vee\text{-in}_{01} \qquad \dfrac{\vdash A \qquad B \vdash A \vee B}{\vdash A \vee B}\ \vee\text{-in}_{10} \qquad \dfrac{\vdash A \qquad \vdash B}{\vdash A \vee B}\ \vee\text{-in}_{11}$$

Using Lemma 9, these rules can be reduced to the following 2 equivalent introduction rules. (We could call ∨-inl also ∨-in$_{\_1}$, but we use a more informative and standard name: "in-left".)

$$\dfrac{\vdash A}{\vdash A \vee B}\ \vee\text{-inl} \qquad \dfrac{\vdash B}{\vdash A \vee B}\ \vee\text{-inr}$$

▶ **Example 11.** Similar to ∨, we can optimize the introduction rules for →. From the truth table we have derived the following 3 intuitionistic introduction rules for →.

$$\frac{A \vdash A \to B \qquad B \vdash A \to B}{\vdash A \to B}\to\text{-in}_{00} \qquad \frac{A \vdash A \to B \qquad \vdash B}{\vdash A \to B}\to\text{-in}_{01} \qquad \frac{\vdash A \qquad \vdash B}{\vdash A \to B}\to\text{-in}_{11}$$

Using Lemma 9, these rules can be reduced to the following 2 equivalent introduction rules.

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

It can easily be shown that the rules $\to$-in$_a$ and $\to$-in$_b$ together are equivalent with the well-known $\to$-in:

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

NB. To derive $\to$-in$_a$ from $\to$-in, one also needs $\to$-el.

As $\to$-in does not conform with our format for rules, we will be using $\to$-in$_a$ and $\to$-in$_b$ as our basic rules and treat $\to$-in as a defined rule, the composition of first $\to$-in$_b$ and then $\to$-in$_a$.

Another optimization we can perform is to replace a rule which has only one case by a rule where the case is the conclusion. To illustrate this: the simplified elimination rules for $\wedge$, $\wedge$-el$_{0\_}$ and $\wedge$-el$_{\_0}$ have only one case. The rule $\wedge$-el$_{0\_}$ can thus be replaced by the rule $\wedge$-ell, which is the usual left projection rule, $\wedge$-elimination-left.

$$\frac{\vdash A \wedge B \qquad A \vdash D}{\vdash D}\wedge\text{-el}_{0\_} \qquad \frac{\vdash A \wedge B}{\vdash A}\wedge\text{-ell}$$

There is a general Lemma stating this simplification is correct.

▶ **Lemma 12.** *A system with a derivation rule of the form to the left is equivalent to the system with this rule replaced by the rule on the right.*

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

Proof. The implication from left to right is immediate. From right to left, assume we have derivations of $\vdash A_1, \ldots, \vdash A_n$. Then, by the rule to the right, we have $\Gamma \vdash B$. Now assume we also have a derivation of $B \vdash D$. By Lemma 5, we also have a derivation of $\Gamma \vdash D$. ◀

▶ **Definition 13.** The *standard derivation rules* for the intuitionistic propositional connectives $\wedge, \vee, \to, \neg, \bot$ and $\top$ are given below. These rules are derived from the truth tables and optimized following Lemmas 9 and 12. We have seen most of the rules in previous Examples, except for the rules for $\top$ and $\bot$, which are derived immediately from Definition 1. The system with these connectives and rules we will call *intuitionistic proposition logic* and if we want to be explicit we write $\Gamma \vdash_i A$ for derivability in this system.

$$\frac{\vdash A \qquad \vdash B}{\vdash A \wedge B}\wedge\text{-in} \qquad \frac{\vdash A \wedge B}{\vdash A}\wedge\text{-ell} \qquad \frac{\vdash A \wedge B}{\vdash B}\wedge\text{-elr}$$

$$\frac{\vdash A}{\vdash A \vee B}\vee\text{-inl} \qquad \frac{\vdash B}{\vdash A \vee B}\vee\text{-inr} \qquad \frac{\vdash A \vee B \quad A \vdash D \quad B \vdash D}{\vdash D}\vee\text{-el}$$

$$\frac{A \vdash A \to B}{\vdash A \to B}\to\text{-in}_a \qquad \frac{\vdash B}{\vdash A \to B}\to\text{-in}_b \qquad \frac{\vdash A \to B \quad \vdash A}{\vdash B}\to\text{-el}$$

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

2.1 Three larger examples

As examples we look in more detail at two ternary connectives and one binary connective. The ternary connectives we treat are if-then-else, the "if-then-else" connective, and most, the ternary connective that is true if at least 2 of the arguments are true. These have been discussed in finer detail in [7], notably the connective if-then-else. The binary connective that we study at the end of this section is the nand, written $A \uparrow B$ for $\mathsf{nand}(A, B)$. It is also known as the *Sheffer stroke*, the well-known connective that is functionally complete classically, where $A \uparrow B$ expresses $\neg(A \wedge B)$.

The truth tables of most and if-then-else are as follows, where we denote if A then B else C by $A{\to}B/C$.

A	B	C	$\mathsf{most}(A, B, C)$	$A{\to}B/C$
0	0	0	0	0
0	0	1	0	1
0	1	0	0	0
0	1	1	1	1
1	0	0	0	0
1	0	1	1	0
1	1	0	1	1
1	1	1	1	1

From the lines in the truth table of $A{\to}B/C$ with a 0 we get the following four elimination rules.

$$\frac{\vdash A{\to}B/C \quad A \vdash D \quad B \vdash D \quad C \vdash D}{\vdash D} \qquad \frac{\vdash A{\to}B/C \quad A \vdash D \quad \vdash B \quad C \vdash D}{\vdash D}$$

$$\frac{\vdash A{\to}B/C \quad \vdash A \quad B \vdash D \quad C \vdash D}{\vdash D} \qquad \frac{\vdash A{\to}B/C \quad \vdash A \quad B \vdash D \quad \vdash C}{\vdash D}$$

Using Lemmas 9 and 12, these can be reduced to the following two. (The two rules on the first line reduce to else-el, the two rules on the second line reduce to then-el.)

$$\frac{\vdash A{\to}B/C \quad A \vdash D \quad C \vdash D}{\vdash D}\text{else-el} \qquad \frac{\vdash A{\to}B/C \quad \vdash A}{\vdash B}\text{then-el}$$

These are not the only possible optimizations: the two rules on the left can also be combined into an "if-el" rule:

$$\frac{\vdash A{\to}B/C \quad B \vdash D \quad C \vdash D}{\vdash D}\ \text{if-el}$$

From the lines in the truth table of $A{\to}B/C$ with a 1 we get the following four introduction rules:

$$\frac{A \vdash A{\to}B/C \quad B \vdash A{\to}B/C \quad \vdash C}{\vdash A{\to}B/C} \qquad \frac{A \vdash A{\to}B/C \quad \vdash B \quad \vdash C}{\vdash A{\to}B/C}$$

$$\frac{\vdash A \quad \vdash B \quad C \vdash A{\to}B/C}{\vdash A{\to}B/C} \qquad \frac{\vdash A \quad \vdash B \quad \vdash C}{\vdash A{\to}B/C}$$

Using Lemmas 9 and 12 can be reduced to the following two. (The two rules on the first line reduce to else-in, the two rules on the second line reduce to then-in.)

$$\boxed{\frac{A \vdash A{\to}B/C \quad \vdash C}{\vdash A{\to}B/C}\ \text{else-in} \qquad \frac{\vdash A \quad \vdash B}{\vdash A{\to}B/C}\ \text{then-in}}$$

Again, these are not the only possible optimizations: the two rules on the right can also be combined into an "if-in" rule:

$$\frac{\vdash B \quad \vdash C}{\vdash A{\to}B/C}\ \text{if-in}$$

In [7], we have studied the if-then-else connective in more detail, and we have shown that if-then-else, together with $\top$ and $\bot$ is *functionally complete*: all other constructive connectives can be defined in terms of it.

From the lines in the truth table of $\mathsf{most}(A,B,C)$ with a 0 we get the following four elimination rules.

$$\frac{\vdash \mathsf{most}(A,B,C) \quad A \vdash D \quad B \vdash D \quad C \vdash D}{\vdash D} \qquad \frac{\vdash \mathsf{most}(A,B,C) \quad A \vdash D \quad B \vdash D \quad \vdash C}{\vdash D}$$

$$\frac{\vdash \mathsf{most}(A,B,C) \quad A \vdash D \quad \vdash B \quad C \vdash D}{\vdash D} \qquad \frac{\vdash \mathsf{most}(A,B,C) \quad \vdash A \quad B \vdash D \quad C \vdash D}{\vdash D}$$

Using Lemmas 9 and 12, these can be reduced to the following three. If we would follow the naming conventions that we introduced earlier, we would have $\mathsf{most\text{-}el}_1 = \mathsf{most\text{-}el}_{00\_}$, $\mathsf{most\text{-}el}_2 = \mathsf{most\text{-}el}_{0\_0}$ and $\mathsf{most\text{-}el}_3 = \mathsf{most\text{-}el}_{\_00}$, but we will not pursue that naming here.

$$\boxed{\begin{array}{ll} \dfrac{\vdash \mathsf{most}(A,B,C) \quad A \vdash D \quad B \vdash D}{\vdash D}\ \mathsf{most\text{-}el}_1 & \dfrac{\vdash \mathsf{most}(A,B,C) \quad A \vdash D \quad C \vdash D}{\vdash B}\ \mathsf{most\text{-}el}_2 \\ \dfrac{\vdash \mathsf{most}(A,B,C) \quad B \vdash D \quad C \vdash D}{\vdash B}\ \mathsf{most\text{-}el}_3 & \end{array}}$$

From the lines in the truth table of $\mathsf{most}(A,B,C)$ with a 1 we get the following four introduction rules:

$$\frac{A \vdash \mathsf{most}(A,B,C) \quad \vdash B \quad \vdash C}{\vdash \mathsf{most}(A,B,C)} \qquad \frac{\vdash A \quad B \vdash \mathsf{most}(A,B,C) \quad \vdash C}{\vdash \mathsf{most}(A,B,C)}$$

$$\frac{\vdash A \quad \vdash B \quad C \vdash \mathsf{most}(A,B,C)}{\vdash \mathsf{most}(A,B,C)} \qquad \frac{\vdash A \quad \vdash B \quad \vdash C}{\vdash \mathsf{most}(A,B,C)}$$

Using Lemmas 9 and 12 can be reduced to the following three.

$$\frac{\vdash A \quad \vdash B}{\vdash \mathsf{most}(A,B,C)}\,\mathsf{most\text{-}in}_1 \qquad \frac{\vdash A \quad \vdash C}{\vdash \mathsf{most}(A,B,C)}\,\mathsf{most\text{-}in}_2 \qquad \frac{\vdash B \quad \vdash C}{\vdash \mathsf{most}(A,B,C)}\,\mathsf{most\text{-}in}_3$$

The truth table for $\mathsf{nand}(A,B)$, which we write as $A \uparrow B$ is as follows.

A	B	$A \uparrow B$
0	0	1
0	1	1
1	0	1
1	1	0

From this we derive the following 3 introduction and 1 elimination rule

$$\frac{A \vdash A \uparrow B \qquad B \vdash A \uparrow B}{\vdash A \uparrow B}\,\uparrow\text{-in}_{00} \qquad \frac{A \vdash A \uparrow B \qquad \vdash B}{\vdash A \uparrow B}\,\uparrow\text{-in}_{01}$$

$$\frac{\vdash A \qquad B \vdash A \uparrow B}{\vdash A \uparrow B}\,\uparrow\text{-in}_{10} \qquad \frac{\vdash A \uparrow B \qquad \vdash A \qquad \vdash B}{\vdash D}\,\uparrow\text{-el}$$

The three introduction rules can be combined to two rules, so our optimized set of deduction rules for nand consists of three rules. We call this nand-logic.

▶ **Definition 14.** The logic with just the connective nand and the three derivation rules below we define as nand-*logic*. We denote derivability in this logic by $\Gamma \vdash_{\uparrow} A$.

$$\frac{A \vdash A \uparrow B}{\vdash A \uparrow B}\,\uparrow\text{-inl} \qquad \frac{B \vdash A \uparrow B}{\vdash A \uparrow B}\,\uparrow\text{-inr} \qquad \frac{\vdash A \uparrow B \quad \vdash A \quad \vdash B}{\vdash D}\,\uparrow\text{-el}$$

We can define the usual connectives of intuitionistic proposition logic (Definition 13) in terms of nand in the usual way. This gives rise to an embedding of intuitionistic proposition logic into the nand-logic.

▶ **Definition 15.**

$$\begin{array}{rcl} \dot\neg A & := & A \uparrow A \\ A \mathrel{\dot\vee} B & := & (A \uparrow A) \uparrow (B \uparrow B) \\ A \mathrel{\dot\wedge} B & := & (A \uparrow B) \uparrow (A \uparrow B) \\ A \mathrel{\dot\to} B & := & A \uparrow (B \uparrow B) \end{array}$$

This gives rise to the following interpretation of intuitionistic proposition logic into nand-logic.

$$\begin{array}{rcl} p^{\uparrow} & := & \dot\neg\dot\neg p \text{ for } p \text{ proposition letter} \\ (\neg A)^{\uparrow} & := & \dot\neg A^{\uparrow} \\ (A \vee B)^{\uparrow} & := & A^{\uparrow} \mathrel{\dot\vee} B^{\uparrow} \\ (A \wedge B)^{\uparrow} & := & A^{\uparrow} \mathrel{\dot\wedge} B^{\uparrow} \\ (A \to B)^{\uparrow} & := & A^{\uparrow} \mathrel{\dot\to} B^{\uparrow} \end{array}$$

This interpretation extends straightforwardly to sets of propositions.

As a side remark, the translation of a proposition letter p could also be chosen to be p in stead of $\dot{\neg}\dot{\neg}p$. Then the soundness statement below (Proposition 17) requires an additional double negation: If $\Gamma \vdash_i A$, then $\Gamma^{\uparrow} \vdash_{\uparrow} \dot{\neg}\dot{\neg}A^{\uparrow}$. The connective $\uparrow$ is very much a "negative connective" and the choice of $\dot{\neg}\dot{\neg}p$ as translation of p renders all formulas $A^{\uparrow}$ negative, so the double negation can be avoided.

Before proving the soundness of the interpretation we give some auxiliary lemmas.

▶ **Lemma 16.** *In* nand*-logic, we have the following.*

1. *For arbitrary propositions A and B,*

   $\dot{\neg}\dot{\neg}(A \uparrow B) \vdash A \uparrow B,$

2. *For every A,*

   $\dot{\neg}\dot{\neg}\dot{\neg}A \vdash \dot{\neg}A.$

3. *For every proposition P from intuitionistic proposition logic,*

   $\dot{\neg}\dot{\neg}P^{\uparrow} \vdash P^{\uparrow}.$

4. *For arbitrary propositions A and B,*

   If $\Gamma, A \vdash B$ *then* $\Gamma, \dot{\neg}B \vdash \dot{\neg}A.$

Proof. The following proves $\dot{\neg}\dot{\neg}(A \uparrow B) \vdash A \uparrow B$. Here $\Gamma = \dot{\neg}\dot{\neg}(A \uparrow B), A, B, A \uparrow B$ and the last $\uparrow$-in rule denotes a successive application of $\uparrow$-inl followed by $\uparrow$-inr. Finally, the lowest $\uparrow$-el has one premise more, which is an exact copy of the derivation of $\dot{\neg}\dot{\neg}(A \uparrow B), A, B \vdash \dot{\neg}(A \uparrow B)$ that is given.

$$\dfrac{\dfrac{\dot{\neg}\dot{\neg}(A \uparrow B), A, B \vdash \dot{\neg}\dot{\neg}(A \uparrow B) \qquad \dfrac{\dfrac{\Gamma \vdash A \uparrow B \qquad \Gamma \vdash A \qquad \Gamma \vdash B}{\dot{\neg}\dot{\neg}(A \uparrow B), A, B, A \uparrow B \vdash \dot{\neg}(A \uparrow B)}\,\uparrow\text{-el}}{\dot{\neg}\dot{\neg}(A \uparrow B), A, B \vdash \dot{\neg}(A \uparrow B)}\,\uparrow\text{-in}}{\dot{\neg}\dot{\neg}(A \uparrow B), A, B \vdash A \uparrow B}\,\uparrow\text{-el}}{\dot{\neg}\dot{\neg}(A \uparrow B) \vdash A \uparrow B}\,\uparrow\text{-in}$$

So, $\dot{\neg}\dot{\neg}\dot{\neg}A \vdash \dot{\neg}A$ follows immediately, and similarly $\dot{\neg}\dot{\neg}P^{\uparrow} \vdash P^{\uparrow}$ for every proposition P from intuitionistic proposition logic.

Now, assuming that $\Gamma, A \vdash B$, we can make the following derivation of $\Gamma, \dot{\neg}B \vdash \dot{\neg}A$, using the fact that $\Gamma, B \uparrow B, A \vdash B$ by weakening.

$$\dfrac{\dfrac{\Gamma, B \uparrow B, A \vdash B \uparrow B \qquad \Gamma, B \uparrow B, A \vdash B \qquad \Gamma, B \uparrow B, A \vdash B}{\Gamma, B \uparrow B, A \vdash A \uparrow A}\,\uparrow\text{-el}}{\Gamma, B \uparrow B \vdash A \uparrow A}\,\uparrow\text{-in}$$

◀

We can now prove the soundness of the interpretation of intuitionistic proposition logic into nand-logic.

▶ **Proposition 17.** *If* $\Gamma \vdash_i A$, *then* $\Gamma^{\uparrow} \vdash_{\uparrow} A^{\uparrow}$.

Proof. The proof is by induction on the derivation of $\Gamma \vdash_i A$, so we have to show that the rules of intuitionistic proposition logic are sound inside nand-logic (after interpretation). We use Lemma 16, notably case (4), which we indicate explicitly in the derivations.

- $\neg$-in: we show that $\neg$-in of Definition 13 is derivable.

$$\dfrac{A \vdash A \uparrow A}{A \vdash A \uparrow A}\,{\uparrow}\text{-in}$$

- $\neg$-el: we show that $\neg$-el of Definition 13 is derivable.

$$\dfrac{\vdash A \uparrow A \qquad \vdash A \qquad \vdash A}{\vdash D}\,{\uparrow}\text{-el}$$

- $\vee$-in: we show that $A \vdash_{\uparrow} A \dot\vee B$ is derivable.

$$\dfrac{\dfrac{A, A \uparrow A \vdash A \uparrow A \qquad A, A \uparrow A \vdash A \qquad A, A \uparrow A \vdash A}{A, A \uparrow A \vdash (A \uparrow A) \uparrow (B \uparrow B)}\,{\uparrow}\text{-el}}{A \vdash (A \uparrow A) \uparrow (B \uparrow B)}\,{\uparrow}\text{-inl}$$

- $\vee$-el: we show that the following rule is derivable (which suffices).

$$\dfrac{\vdash A \dot\vee B \qquad A \vdash D \qquad B \vdash D}{\vdash \dot\neg\dot\neg D}$$

$$\dfrac{\dfrac{\vdash (A \uparrow A) \uparrow (B \uparrow B) \qquad \dfrac{A \vdash D}{D \uparrow D \vdash A \uparrow A}\,16(4) \qquad \dfrac{B \vdash D}{D \uparrow D \vdash B \uparrow B}\,16(4)}{D \uparrow D \vdash (D \uparrow D) \uparrow (D \uparrow D)}\,{\uparrow}\text{-el}}{\vdash (D \uparrow D) \uparrow (D \uparrow D)}\,{\uparrow}\text{-inl}$$

- $\wedge$-el: we show that $A \dot\wedge B \vdash_{\uparrow} \dot\neg\dot\neg A$ is derivable.

$$\dfrac{\dfrac{\dfrac{A \dot\wedge B \vdash (A \uparrow B) \uparrow (A \uparrow B) \qquad \dfrac{\dfrac{A \uparrow A \vdash A \uparrow A \quad A \vdash A}{A \uparrow A, A \vdash A \uparrow B}\,{\uparrow}\text{-el}}{A \uparrow A \vdash A \uparrow B}\,{\uparrow}\text{-inl}}{A \dot\wedge B, A \uparrow A \vdash A}\,{\uparrow}\text{-el}}{A \dot\wedge B, A \uparrow A \vdash (A \uparrow A) \uparrow (A \uparrow A)}\,16(4)}{A \dot\wedge B \vdash (A \uparrow A) \uparrow (A \uparrow A)}\,{\uparrow}\text{-inl}$$

- $\wedge$-in: we show that the following rule is derivable (which suffices).

$$\dfrac{\vdash A \qquad \vdash B}{\vdash A \dot\wedge B}$$

$$\dfrac{\dfrac{A \uparrow B \vdash A \uparrow B \qquad \vdash A \qquad \vdash B}{A \uparrow B \vdash (A \uparrow B) \uparrow (A \uparrow B)}\,{\uparrow}\text{-el}}{\vdash (A \uparrow B) \uparrow (A \uparrow B)}\,{\uparrow}\text{-inl}$$

- $\to$-in: we show that the following rule is derivable (which suffices).

$$\dfrac{A \vdash B}{\vdash A \dot\to B}$$

$$\dfrac{\dfrac{\dfrac{B \uparrow B \vdash B \uparrow B \qquad A \vdash B \qquad A \vdash B}{A, B \uparrow B \vdash A \uparrow (B \uparrow B)}\,{\uparrow}\text{-el}}{A \vdash A \uparrow (B \uparrow B)}\,{\uparrow}\text{-inr}}{\vdash A \uparrow (B \uparrow B)}\,{\uparrow}\text{-inl}$$

- $\to$-el: we show that the following rule is derivable (which suffices).

$$\frac{\vdash A \mathbin{\dot{\to}} B \qquad \vdash A}{\vdash \dot{\neg}\dot{\neg} B}$$

$$\cfrac{\cfrac{\cfrac{\vdash A \uparrow (B \uparrow B) \qquad \vdash A \qquad B \uparrow B \vdash B \uparrow B}{B \uparrow B \vdash B}\ \uparrow\text{-el}}{B \uparrow B \vdash (B \uparrow B) \uparrow (B \uparrow B)}\ 16(4)}{\vdash (B \uparrow B) \uparrow (B \uparrow B)}\ \uparrow\text{-inl}$$ ◀

The reverse of Proposition 17 does not hold. For example, $\nvdash p \vee \neg p$, for p a proposition letter, while $(p \vee \neg p)^{\uparrow} = (\dot{p} \uparrow \dot{p}) \uparrow (\dot{\neg}\dot{p} \uparrow \dot{\neg}\dot{p})$, where $\dot{p} := \dot{\neg}\dot{\neg}p$. The proposition $(A \uparrow A) \uparrow (\dot{\neg}A \uparrow \dot{\neg}A)$ is derivable in nand-logic for any A (note that $\dot{\neg}A = A \uparrow A$):

$$\cfrac{\cfrac{\dot{\neg}A \uparrow \dot{\neg}A \vdash \dot{\neg}A \uparrow \dot{\neg}A \qquad A \uparrow A \vdash \dot{\neg}A \qquad A \uparrow A \vdash \dot{\neg}A}{A \uparrow A, \dot{\neg}A \uparrow \dot{\neg}A \vdash (A \uparrow A) \uparrow (\dot{\neg}A \uparrow \dot{\neg}A)}\ \uparrow\text{-el}}{\vdash (A \uparrow A) \uparrow (\dot{\neg}A \uparrow \dot{\neg}A)}\ \uparrow\text{-in}$$

There is also an obvious mapping from nand-logic to intuitionistic proposition logic, by interpreting $A \uparrow B$ as $\neg(A \wedge B)$. As a matter of fact, it can also be shown in the joint system (i.e. where we add nand to intuitionistic proposition logic) that $A \uparrow B$ and $\neg(A \wedge B)$ are equivalent: $A \uparrow B \vdash \neg(A \wedge B)$ and $\neg(A \wedge B) \vdash A \uparrow B$. In presence of the implication and conjunction connective, the latter can be reformulated as $\vdash A \uparrow B \longleftrightarrow \neg(A \wedge B)$ (where, as usual, we let $C \longleftrightarrow D$ abbreviate $(C \to D) \wedge (D \to C)$).

▶ **Definition 18.** We define the mapping $(-)^{\downarrow}$ from nand-logic to intuitionistic proposition logic by defining

$$(A \uparrow B)^{\downarrow} := \neg(A^{\downarrow} \wedge B^{\downarrow})$$

and further by induction on propositions. This mapping extends to sets of hypotheses Γ in the obvious way.

▶ **Proposition 19.** *If $\Gamma \vdash_{\uparrow} A$, then $\Gamma^{\downarrow} \vdash_i A^{\downarrow}$.*

Proof. By induction on the derivation. The only thing to show is that the rules $\uparrow$-el, $\uparrow$-inl and $\uparrow$-inr are sound in intuitionistic proposition logic is we interpret $A \uparrow B$ as $\neg(A \wedge B)$. So we have to verify the soundness of the following rules.

$$\frac{A \vdash \neg(A \wedge B)}{\vdash \neg(A \wedge B)} \qquad \frac{B \vdash \neg(A \wedge B)}{\vdash \neg(A \wedge B)} \qquad \frac{\vdash \neg(A \wedge B) \quad \vdash A \quad \vdash B}{\vdash D}$$

A simple inspection shows that these rules are sound in intuitionistic proposition logic. ◀

We can now formulate a Glivenko-like theorem that relates nand-logic and intuitionistic proposition logic. (Glivenko's theorem, e.g. see [22], relates intuitionistic and classical proposition logic via the double negation.)

▶ **Proposition 20.** *For A a proposition of intuitionistic proposition logic,*

$$\vdash_i A^{\uparrow\downarrow} \longleftrightarrow \neg\neg A$$

.

Proof. By induction on the structure of A.

- $A = p$, a proposition letter. Then $p^{\uparrow\downarrow} = (\dot{\neg}\dot{\neg}p)^{\downarrow} = \neg(\neg(p \wedge p) \wedge \neg(p \wedge p)) \longleftrightarrow \neg\neg p$.
- $A = \neg B$. Then $(\neg B)^{\uparrow\downarrow} = (B \uparrow B)^{\downarrow} = \neg(B \wedge B) \longleftrightarrow \neg\neg\neg B$.
- $A = B \vee C$. Then $(B \vee C)^{\uparrow\downarrow} = ((B \uparrow B) \uparrow (C \uparrow C))^{\downarrow} = \neg(\neg(B \wedge B) \wedge \neg(C \wedge C)) \longleftrightarrow \neg\neg(B \vee C)$.
  For the equivalence $\neg(\neg B \wedge \neg C) \longleftrightarrow \neg\neg(B \vee C)$: from left to right, if $\neg(B \vee C)$, then $\neg B$ and $\neg C$, so we have a contradiction with $\neg(\neg B \wedge \neg C)$; from right to left, if $\neg B \wedge \neg C$, then $\neg B$ and so from $B \vee C$ we derive C, contradiction, so we derive $\neg(B \vee C)$, but this contradicts $\neg\neg(B \vee C)$, so we conclude that $\neg(\neg B \wedge \neg C)$
- $A = B \wedge C$. Then $(B \wedge C)^{\uparrow\downarrow} = ((B \uparrow C) \uparrow (B \uparrow C))^{\downarrow} = \neg(\neg(B \wedge C) \wedge \neg(B \wedge C)) \longleftrightarrow \neg\neg(B \wedge C)$.
- $A = B \to C$. Then $(B \to C)^{\uparrow\downarrow} = (B \uparrow (C \uparrow C))^{\downarrow} = \neg(B \wedge \neg(C \wedge C)) \longleftrightarrow \neg\neg(B \to C)$.
  For the equivalence $\neg(B \wedge \neg C) \longleftrightarrow \neg\neg(B \to C)$: From left to right, assume $\neg(B \to C)$; if C, then $B \to C$, so from $\neg(B \to C)$ we get $\neg C$; then if B we also have $B \wedge \neg C$, contradicting $\neg(B \wedge \neg C)$, so we have $\neg B$; but from $\neg B$ we get $B \to C$. Contradiction, so we conclude $\neg\neg(B \to C)$. From right to left: Assume $B \wedge \neg C$. Then $B \to C$ implies C, contradiction, so $\neg(B \to C)$, contradicting $\neg(B \to C)$, so we conclude $\neg(B \wedge \neg C)$. ◀

▶ **Corollary 21.** *For A a proposition in intuitionistic proposition logic,*

$$\vdash_i \neg\neg A \qquad \Longleftrightarrow \qquad \vdash_\uparrow A^\uparrow.$$

Proof. If $\vdash_i \neg\neg A$, then $\vdash_\uparrow \dot{\neg}\dot{\neg}A^\uparrow$ by Proposition 17, and so $\vdash_\uparrow \dot{\neg}\dot{\neg}A^\uparrow$ by Lemma 16(1).
If $\vdash_\uparrow A^\uparrow$, then $\vdash_i A^{\uparrow\downarrow}$ by Proposition 19, so $\vdash_i \neg\neg A$ by Proposition 20. ◀

3 Convertibilities and conversion

The notion of *detour convertibility* has already been described in [7]: an introduction of Φ immediately followed by an elimination of Φ. (In [7] it was called *direct cut* but – although the literature is not completely consistent on this point – the notion of cut is usually reserved for sequent calculus and for natural deduction one uses the terminology of convertibility.) In such case there is (referring back to the truth table, see Definition 1) at least one k for which $a_k \neq b_k$. In case $a_k = 0, b_k = 1$, we have a sub-derivation Σ of $\vdash A_k$ and a sub-derivation Θ of $A_k \vdash D$ and we can plug Σ on top of Θ to obtain a derivation of $\vdash D$. In case $a_k = 1, b_k = 0$, we have a sub-derivation Σ of $A_k \vdash \Phi$ and a sub-derivation Θ of $\vdash A_k$ and we can plug Θ on top of Σ to obtain a derivation of $\vdash \Phi$. This is then used as a hypothesis for the elimination rule (that remains in this case) instead of the original one that was a consequence of the introduction rule (that now disappears).

In general there are more k for which $a_k \neq b_k$, so the general detour conversion procedure is non-deterministic. We view this non-determinism as a natural feature in natural deduction; the fact that for some connectives (or combination of connectives), detour conversion is deterministic is an "emerging" property. We will show examples of the non-determinism of detour conversion later.

The introduction of a formula Φ immediately followed by an elimination of Φ we will call a *detour convertibility*. In general in between the introduction rule for Φ and the elimination rule for Φ, there may be other auxiliary rules, so occasionally we may have to first permute the elimination rule with these auxiliary rules to obtain a detour convertibility that can be reduced away. So, we will also define the notion of *permutation convertibility* and of *permutation conversion*.

▶ **Definition 22.** Let c be a connective of arity n, with an elimination rule and an intuitionistic introduction rule derived from the truth table, as in Definition 1. So suppose we have the following rules in the truth table t_c.

p_1	$\ldots$	p_n	$c(p_1,\ldots,p_n)$
a_1	$\ldots$	a_n	0
b_1	$\ldots$	b_n	1

A *detour convertibility* in a derivation is a pattern of the following form, where $\Phi = c(A_1,\ldots,A_n)$.

$$\dfrac{\dfrac{\cdots \quad \begin{array}{c}\boxed{\Sigma_j}\\ \Gamma\vdash A_j\end{array} \quad \cdots \qquad \cdots \quad \begin{array}{c}\boxed{\Sigma_i}\\ \Gamma, A_i\vdash\Phi\end{array}\quad \cdots}{\Gamma\vdash\Phi}\,\text{in} \qquad \cdots \quad \begin{array}{c}\boxed{\Pi_k}\\ \Gamma\vdash A_k\end{array} \quad \cdots \qquad \cdots \quad \begin{array}{c}\boxed{\Pi_\ell}\\ \Gamma, A_\ell\vdash D\end{array}\quad \cdots}{\Gamma\vdash D}\,\text{el}$$

- Here, in is an arbitrary introduction rule. In this rule, A_j ranges over all propositions where $b_j = 1$; A_i ranges over all propositions where $b_i = 0$,
- Here, el is an arbitrary elimination rule. In this rule, A_k ranges over all propositions where $a_k = 1$; A_ℓ over all propositions where $a_\ell = 0$,

A *detour conversion* is defined by replacing the derivation pattern above by

1. If $\ell = j$ for some ℓ, j (that is: $A_\ell = A_j$):

$$\begin{array}{c}\begin{array}{ccc}\vdots\,\boxed{\Sigma_j} & & \vdots\,\boxed{\Sigma_j}\\ \Gamma\vdash A_j & \ldots & \Gamma\vdash A_j\end{array}\\ \vdots\,\boxed{\Pi_\ell}\\ \Gamma\vdash D\end{array}$$

2. If $k = i$ for some k, i (that is: $A_k = A_i$):

$$\dfrac{\begin{array}{c}\begin{array}{ccc}\vdots\,\boxed{\Pi_k} & & \vdots\,\boxed{\Pi_k}\\ \Gamma\vdash A_i & \ldots & \Gamma\vdash A_i\end{array}\\ \vdots\,\boxed{\Sigma_i}\\ \Gamma\vdash\Phi\end{array} \qquad \cdots \quad \begin{array}{c}\vdots\,\boxed{\Pi_k}\\ \Gamma\vdash A_k\end{array} \quad \cdots \qquad \cdots \quad \begin{array}{c}\vdots\,\boxed{\Pi_\ell}\\ \Gamma, A_\ell\vdash D\end{array}\quad \cdots}{\Gamma\vdash D}\,\text{el}$$

There may be several choices for the i and j in the previous definition, so detour elimination is non-deterministic in general. We give an example of most to illustrate this. For simplicity, we use the optimized rules.

▶ **Example 23.** Consider the following detour convertibility for most.

$$\dfrac{\dfrac{\begin{array}{c}\vdots\,\boxed{\Sigma_1}\\ \Gamma\vdash A\end{array} \qquad \begin{array}{c}\vdots\,\boxed{\Sigma_2}\\ \Gamma\vdash B\end{array}}{\Gamma\vdash \mathsf{most}(A,B,C)}\,\mathsf{most\text{-}in}_1 \qquad \begin{array}{c}\vdots\,\boxed{\Pi_1}\\ \Gamma, A\vdash D\end{array} \qquad \begin{array}{c}\vdots\,\boxed{\Pi_2}\\ \Gamma, B\vdash D\end{array}}{\Gamma\vdash D}\,\mathsf{most\text{-}el}_1$$

Here we can reduce to either one of the following derivations of $\Gamma\vdash D$, which shows that the detour conversion process is not Church-Rosser. (Of course, one could fix a choice, e.g.

always take the first possible detour convertibility from the left, but that would be completely arbitrary.)

$$\begin{array}{c} \vdots\,\boxed{\Sigma_1} \qquad\quad \vdots\,\boxed{\Sigma_1} \\ \Gamma \vdash A \quad \ldots \quad \Gamma \vdash A \\ \vdots\,\boxed{\Pi_1} \\ \Gamma \vdash D \end{array} \qquad\qquad \begin{array}{c} \vdots\,\boxed{\Sigma_2} \qquad\quad \vdots\,\boxed{\Sigma_2} \\ \Gamma \vdash B \quad \ldots \quad \Gamma \vdash B \\ \vdots\,\boxed{\Pi_2} \\ \Gamma \vdash D \end{array}$$

A more concrete example is the following.

$$\dfrac{\dfrac{\dfrac{A\wedge B\vdash A\wedge B}{A\wedge B\vdash A}\wedge\text{-ell} \qquad \dfrac{A\wedge B\vdash A\wedge B}{A\wedge B\vdash B}\wedge\text{-elr}}{A\wedge B\vdash \mathsf{most}(A,B,C)}\mathsf{most}\text{-in}_1 \qquad \dfrac{A\vdash A}{A\vdash A\vee B}\vee\text{-inl} \qquad \dfrac{B\vdash B}{B\vdash A\vee B}\vee\text{-inr}}{A\wedge B\vdash A\vee B}\mathsf{most}\text{-el}_1$$

This derivation can either be reduced to a derivation of $A \wedge B \vdash A \vee B$ via $A \wedge B \vdash A$ or via $A \wedge B \vdash B$.

It can happen that the introduction of a formula $\Phi = c(A_1, \ldots, A_n)$ is not followed directly by an elimination for c, but first by other elimination rules, where Φ acts as a minor premise. In that way, a detour convertibility can be "blocked" by other elimination rules. So, apart from the detour conversion elimination arising from an introduction rule immediately followed by an elimination, we have a notion of "hidden" or *permutation convertibility*, where we want to permute one elimination rule over another.

▶ **Example 24.**

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

In this derivation, the detour convertibility arising from $\to$-in$_a$ followed by $\to$-el is blocked by the $\vee$-el rule where the major premise of the $\to$-el rule is a minor premise. This is a *permutation convertibility*, which can be contracted by permuting the $\to$-el rule over the $\vee$-el rule.

▶ **Definition 25.** Let c and c' be connectives of arity n and n', with elimination rules r and r' respectively, both derived from the truth table. A *permutation convertibility* in a derivation is a pattern of the following form, where $\Phi = c(B_1, \ldots, B_n)$, $\Psi = c'(A_1, \ldots, A_{n'})$.

$$\dfrac{\dfrac{\Gamma\vdash\Psi\ldots \quad \begin{array}{c}\vdots\,\boxed{\Sigma_j}\\ \Gamma\vdash A_j\end{array} \quad \ldots \quad \ldots \quad \begin{array}{c}\vdots\,\boxed{\Sigma_i}\\ \Gamma,A_i\vdash\Phi\end{array} \quad \ldots}{\Gamma\vdash\Phi}\text{el}_{r'} \qquad \ldots \quad \begin{array}{c}\vdots\,\boxed{\Pi_k}\\ \Gamma\vdash B_k\end{array} \quad \ldots \quad \ldots \quad \begin{array}{c}\vdots\,\boxed{\Pi_\ell}\\ \Gamma,B_\ell\vdash D\end{array} \quad \ldots}{\Gamma\vdash D}\text{el}_r$$

- A_j ranges over all propositions that have a 1 in the truth table of c'; A_i ranges over all propositions that have a 0,
- B_k ranges over all propositions that have a 1 in the truth table of c; B_ℓ ranges over all propositions that have a 0.

The *permutation conversion* is defined by replacing the derivation pattern above by

$$\dfrac{\Gamma \vdash \Psi \ldots \quad \begin{array}{c}\vdots\ \boxed{\Sigma_j}\\ \Gamma \vdash A_j\end{array} \quad \ldots \quad \ldots \quad \dfrac{\begin{array}{c}\vdots\ \boxed{\Sigma_i}\\ \Gamma, A_i \vdash \Phi\end{array} \quad \ldots \quad \begin{array}{c}\vdots\ \boxed{\Pi_k}\\ \Gamma, A_i \vdash B_k\end{array} \quad \ldots \quad \ldots \quad \begin{array}{c}\vdots\ \boxed{\Pi_\ell}\\ \Gamma, A_i, B_\ell \vdash D\end{array} \quad \ldots}{\Gamma, A_i \vdash D}\,\mathrm{el}_r}{\Gamma \vdash D}\,\mathrm{el}_{r'}$$

This gives rise to copying of sub-derivations: for every A_i we copy the sub-derivations $\Pi_1, \ldots, \Pi_n$.

NB. Due to weakening, $\boxed{\Pi_k}$ is also a derivation of $\Gamma, A_i \vdash B_k$ and $\boxed{\Pi_\ell}$ is also a derivation of $\Gamma, A_i, B_\ell \vdash D$.

▶ **Example 26.** If we reduce the permutation convertibility in Example 24, we obtain the following derivation.

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

4 The Curry-Howard isomorphism

We now define typed proof-terms for derivations, which enables the study of "proofs as terms" and emphasis es the computational interpretation of proofs, as detour conversion and permutation conversion will correspond to reductions on these proof-terms. For each connective c we give a general definition of proof-terms for the full set of derivation rules for c, as they have been derived from the truth table. This amounts to a system $\lambda^{\mathcal{C}}$, parametrized by a set of connectives $\mathcal{C}$. Then, to clarify the approach, we show how this works out on a number of examples.

Often, we don't want to consider the full rules for a connective c, but only the optimized rules, following Lemmas 9 and 12. For these optimized rules, there is also a straightforward definition of proof-terms and of the reduction relation associated with (detour, permutation) conversion. In the next Section 5 we show in detail how Lemmas 9 and 12 can be extended to terms and reductions: the proof-terms for the optimized rules can be defined in terms of our original calculus $\lambda^{\mathcal{C}}$, and the reduction rules for the optimized proof terms are an instance of reductions in the original calculus (often multi-step).

▶ **Definition 27.** Given a logic with intuitionistic derivation rules, as derived from truth tables for a set of connectives $\mathcal{C}$, as in Definition 1, we now define the typed λ-calculus $\lambda^{\mathcal{C}}$. The system $\lambda^{\mathcal{C}}$ has judgments $\Gamma \vdash t : A$, where A is a formula, Γ is a set of declarations $\{x_1 : A_1, \ldots, x_m : A_m\}$, where the A_i are formulas and the x_i are term-variables such that every x_i occurs at most once in Γ, and t is a *proof-term.*

Let $c \in \mathcal{C}$ be a connective of arity n, which has 2^n rules (introduction plus elimination rules). For each rule r we have a term: an *introduction term*, $\{\overline{p}\,;\,\overline{Q}\}_r$, if r is an introduction rule, or an *elimination term*, $t \cdot_r [\overline{p}\,;\,\overline{Q}]$, if r is an elimination rule. Here, t is again a term, $\overline{p}$ is a finite sequence of terms and $\overline{Q}$ is a finite sequence of *abstracted terms* $\lambda x : A.q$, where x is a term-variable, A is a proposition and q is a term. So the abstract syntax for proof-terms, Term, is as follows.

$$t ::= x \mid \{\overline{t}\,;\,\overline{\lambda x : A.t}\}_r \mid t \cdot_r [\overline{t}\,;\,\overline{\lambda x : A.t}]$$

where x ranges over variables and r ranges over the rules of all the connectives.

The terms are *typed* using the following derivation rules.

$$\frac{}{\Gamma \vdash x_i : A_i} \text{ if } x_i : A_i \in \Gamma$$

$$\frac{\ldots \Gamma \vdash p_j : A_j \ldots \quad \ldots \Gamma, y_i : A_i \vdash q_i : \Phi \ldots}{\Gamma \vdash \{\overline{p} \; ; \; \overline{\lambda y : A.q}\}_r : \Phi} \text{ in}$$

$$\frac{\Gamma \vdash t : \Phi \quad \ldots \Gamma \vdash p_k : A_k \ldots \quad \ldots \Gamma, y_\ell : A_\ell \vdash q_\ell : D}{\Gamma \vdash t \cdot_r [\overline{p} \; ; \; \overline{\lambda y : A.q}] : D} \text{ el}$$

Here, $\overline{p}$ is the sequence of terms $p_1, \ldots, p_{m'}$ for all the 1-entries in the truth table, and $\overline{\lambda y : A.q}$ is the sequence of terms $\lambda y_1 : A_1.q_1, \ldots, \lambda y_m : A_m.q_m$ for all the 0-entries in the truth table.

▶ **Convention 28.** *We view the λ-abstracted variables as being* typed *so we write $\overline{\lambda y : A.q}$ and $\lambda y_1 : A_1.q_1, \ldots, \lambda y_m : A_r.q_m$. However, these types clutter up the syntax considerably, so in practice we will almost always leave the types implicit. In case we want to stress that a variable has a certain type, or in case type information enhances the understanding, we will write the type as a superscript, so $\lambda x^A.p$ in stead of $\lambda x : A.p$.*

We will sometimes leave the rule r that the elimination or introduction term corresponds to implicit, or we will just number the terms or introduce special names for them without explicit reference to the rule. It should be clear that every line in the truth table for the connective gives rise to one rule, which again gives rise to one term-constructor, which is either an elimination or an introduction term-constructor.

There are term reduction rules that correspond to detour conversion.

▶ **Definition 29.** Given a detour convertibility as defined in Definition 22, we add reduction rules for the associated terms as follows.

- For the $\ell = j$ case, that is, $y_\ell : A_\ell$ and $p_j : A_j$ with $A_\ell = A_j$:

$$\{\overline{p, p_j} \; ; \; \overline{\lambda x.q}\} \cdot [\overline{s} \; ; \; \overline{\lambda y.r, \lambda y_\ell.r_\ell}] \longrightarrow_a r_\ell[y_\ell := p_j]$$

- For the $k = i$ case, that is, $s_k : A_k$ and $x_i : A_i$ with $A_k = A_i$:

$$\{\overline{p} \; ; \; \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot [\overline{s, s_k} \; ; \; \overline{\lambda y.r}] \longrightarrow_a q_i[x_i := s_k] \cdot [\overline{s, s_k} \; ; \; \overline{\lambda y.r}]$$

For simplicity of presentation we write the "matching cases" in Definition 22 as last term of the sequence. So when writing $\overline{p, p_j}$, this should be understood as a sequence of terms $p_1, \ldots, p_j, \ldots p_{m'}$, where we have singled out the p_j that matches the r_ℓ in $\overline{\lambda y.r, \lambda y_\ell.r_\ell}$. Similarly for $\overline{s, s_k}$ and $\overline{\lambda x.q, \lambda x_i.q_i}$.

It is important to note that there is always (at least one) "matching case", because introduction rules and elimination rules comes from different lines in the truth table.

The reduction is extended in the straightforward way to sub-terms, by defining it as a congruence with respect to the term constructions.

This Definition gives a reduction rule, and possibly more than one, for every combination of an elimination and an introduction. For an n-ary connective, there are 2^n rules in the truth table, and therefore 2^n term-constructors (introduction plus elimination constructors). We now give the examples of the proof-terms for $\vee$ and $\wedge$ in full. In the rules we will always omit the context Γ.

▶ **Example 30.** The rules for disjunction are as follows.

$$\frac{\vdash t : A \vee B \quad x : A \vdash p : D \quad y : B \vdash q : D}{\vdash t \bullet^{\vee} [\,;\, \lambda x.p, \lambda y.q] : D} \qquad \frac{z : A \vdash r : A \vee B \quad \vdash b : B}{\vdash \{b \,;\, \lambda z.r\}^{\vee}_{1} : A \vee B}$$

$$\frac{\vdash a : A \quad z : B \vdash r : A \vee B}{\vdash \{a \,;\, \lambda z.r\}^{\vee}_{2} : A \vee B} \qquad \frac{\vdash a : A \quad \vdash b : B}{\vdash \{a, b \,;\, \}^{\vee}_{3} : A \vee B}$$

We could have followed our earlier introduced naming convention and index the operators with the line of the truth table they arise from. Then we would write $\{b \,;\, \lambda z.r\}^{\vee}_{01}$ for $\{b \,;\, \lambda z.r\}^{\vee}_{1}$, $\{a \,;\, \lambda z.r\}^{\vee}_{10}$ for $\{a \,;\, \lambda z.r\}^{\vee}_{2}$ and $\{a, b \,;\, \}^{\vee}_{11}$ $\{a, b \,;\, \}^{\vee}_{3}$. This easily clutters up notation, so we don't pursue that.

The reduction rules are

$$\begin{array}{rcl}
\{b \,;\, \lambda z.r\}^{\vee}_{1} \bullet^{\vee} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & q[y := b] \\
\{a \,;\, \lambda z.r\}^{\vee}_{2} \bullet^{\vee} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & p[x := a] \\
\{a, b \,;\, \}^{\vee}_{3} \bullet^{\vee} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & p[x := a] \\
\{a, b \,;\, \}^{\vee}_{3} \bullet^{\vee} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & q[y := b]
\end{array}$$

From the last two cases, we see that the Church-Rosser property (confluence) is lost.

The rules for conjunction are as follows.

$$\frac{\vdash t : A \wedge B \quad x : A \vdash p : D \quad y : B \vdash q : D}{\vdash t \bullet^{\wedge}_{1} [\,;\, \lambda x.p, \lambda y.q] : D} \qquad \frac{\vdash t : A \wedge B \quad \vdash a : A \quad y : B \vdash q : D}{\vdash t \bullet^{\wedge}_{2} [a \,;\, \lambda y.q] : D}$$

$$\frac{\vdash t : A \wedge B \quad x : A \vdash p : D \quad \vdash b : B}{\vdash t \bullet^{\wedge}_{3} [b \,;\, \lambda x.p] : D} \qquad \frac{\vdash a : A \quad \vdash b : B}{\vdash \{a, b \,;\, \}^{\wedge} : A \wedge B}$$

The reduction rules are

$$\begin{array}{rcl}
\{a, b \,;\, \}^{\wedge} \bullet^{\wedge}_{1} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & p[x := a] \\
\{a, b \,;\, \}^{\wedge} \bullet^{\wedge}_{1} [\,;\, \lambda x.p, \lambda y.q] & \longrightarrow_a & q[y := b] \\
\{a, b \,;\, \}^{\wedge} \bullet^{\wedge}_{2} [a' \,;\, \lambda y.q] & \longrightarrow_a & q[y := b] \\
\{a, b \,;\, \}^{\wedge} \bullet^{\wedge}_{3} [b' \,;\, \lambda x.p] & \longrightarrow_a & p[x := a]
\end{array}$$

From the first two cases, we see that the Church-Rosser property (confluence) is lost.

In Example 39 we will show how we can define proof-terms for the optimized rules for $\wedge$ in terms of the proof-terms for the full rules, while preserving reduction.

In the reduction for the terms for $\vee$ and $\wedge$, an elimination is always removed at each step. The situation gets more interesting with implication.

▶ **Example 31.** The rules for implication are as follows.

$$\frac{x : A \vdash p : A \to B \quad y : B \vdash q : A \to B}{\vdash \{\,;\, \lambda x.p, \lambda y.q\}^{\to}_{1} : A \to B} \qquad \frac{x : A \vdash p : A \to B \quad \vdash b : B}{\vdash \{b \,;\, \lambda x.p\}^{\to}_{2} : A \to B}$$

$$\frac{\vdash t : A \to B \quad \vdash a : A \quad z : B \vdash r : D}{\vdash t \bullet^{\to} [a \,;\, \lambda z.r] : D} \qquad \frac{\vdash a : A \quad \vdash b : B}{\vdash \{a, b \,;\, \}^{\to}_{3} : A \to B}$$

The reduction rules are

$$\begin{array}{rcl}
\{\ ;\ \lambda x.p, \lambda y.q\}^{\to}_1 \cdot^{\to} [a\ ;\ \lambda z.r] & \longrightarrow_a & p[x := a] \cdot^{\to} [a\ ;\ \lambda z.r] \\
\{b\ ;\ \lambda x.p\}^{\to}_2 \cdot^{\to} [a\ ;\ \lambda z.r] & \longrightarrow_a & r[z := b] \\
\{b\ ;\ \lambda x.p\}^{\to}_2 \cdot^{\to} [a\ ;\ \lambda z.r] & \longrightarrow_a & p[x := a] \cdot^{\to} [a\ ;\ \lambda z.r] \\
\{a', b\ ;\ \}^{\to}_3 \cdot^{\to} [a\ ;\ \lambda z.r] & \longrightarrow_a & r[z := b]
\end{array}$$

From the second and third case, we can see that Church-Rosser is lost. In the first and the third case, we see that the elimination remains.

In Example 41 we will show how we can define proof-terms for the optimized rules for $\to$ in terms of the proof-terms for the full rules, while preserving reduction. In Definition 48 we will define the standard rules for $\to$.

We now extend the reduction on proof-terms to also capture the permutation conversions of Definition 25. This gives rise to two elimination constructs permuting with each other.

▶ **Definition 32.** Given a permutation convertibility as defined in Definition 25, we add reduction rules for the associated terms as follows.

$$(t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}]) \cdot [\overline{s}\ ;\ \overline{\lambda y.r}] \quad \longrightarrow_b \quad t \cdot [\overline{p}\ ;\ \overline{\lambda x.(q \cdot [\overline{s}\ ;\ \overline{\lambda y.r}])}]$$

Here, the notation $\overline{\lambda x.(q \cdot [\overline{s}\ ;\ \overline{\lambda y.r}])}$ should be understood as a sequence $\lambda x_1.q_1, \ldots, \lambda x_m.q_m$ where each q_j is replaced by $q_j \cdot [\overline{s}\ ;\ \overline{\lambda y.r}]$.

The reduction is extended in the straightforward way to sub-terms, by defining it as a congruence with respect to the term constructions.

▶ **Notation 33.** *We omit brackets by letting the application operator* $- \cdot -$ *associate to the left, so* $t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}] \cdot [\overline{s}\ ;\ \overline{\lambda y.r}]$ *denotes* $(t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}]) \cdot [\overline{s}\ ;\ \overline{\lambda y.r}]$*. We will also omit the brackets in* $\overline{\lambda x.(q \cdot [\overline{s}\ ;\ \overline{\lambda y.r}])}$*, because no ambiguity can arise here.*

We treat the well-known example from intuitionistic logic of the $\vee$-elimination, where a permutation convertibility can occur. See also Example 24.

▶ **Example 34.**

$$\dfrac{\dfrac{\vdash t : A \vee B \quad x : A \vdash p : C \to D \quad y : B \vdash q : C \to D}{\vdash t \cdot^{\vee} [\ ;\ \lambda x.p, \lambda y.q] : C \to D} \qquad \vdash c : C \quad z : D \vdash r : E}{\vdash t \cdot^{\vee} [\ ;\ \lambda x.p, \lambda y.q] \cdot^{\to} [c\ ;\ \lambda z.r] : E}$$

We observe two consecutive elimination rules, where a potential detour convertibility, arising e.g. when q is an introduction term, is blocked by the $\vee$-elimination.

The term reduces as follows

$$t \cdot^{\vee} [\ ;\ \lambda x.p, \lambda y.q] \cdot^{\to} [c\ ;\ \lambda z.r] \longrightarrow_b t \cdot^{\vee} [\ ;\ \lambda x.p \cdot^{\to} [c\ ;\ \lambda z.r], \lambda y.q \cdot^{\to} [c\ ;\ \lambda z.r]]$$

We can now easily define the terms in normal-form under the combined reduction $\longrightarrow_{ab}$. The proof is straightforward and comes from the fact that an introduction followed by an elimination is always a redex. (There is always a "matching case" in Definition 29.)

▶ **Lemma 35.** *The set of* terms in normal form *of* $\mathsf{IPC}_{\mathcal{C}}$*,* NF*is characterized by the following inductive definition.*

- $x \in \mathsf{NF}$ *for every variable* x*,*
- $\{\overline{p}\ ;\ \overline{\lambda y.q}\} \in \mathsf{NF}$ *if all* p_i *and* q_j *are in* NF*,*
- $x \cdot [\overline{p}\ ;\ \overline{\lambda y.q}] \in \mathsf{NF}$ *if all* p_i *and* q_j *are in* NF *and* x *is a variable.*

▶ Remark. In [23], yet another notion of convertibility is defined, called *simplification convertibility*. This is a situation where the assumption is unused in an introduction or elimination rule and the rule can be removed all together. Adding these rules is not necessary for the sub-formula property, so we don't introduce it here. On the term level, an elimination of simplification convertibilities would amount to the following reduction rules.

$$\begin{array}{rcll} t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}] & \longrightarrow & q_i & \text{if } x_i \notin \mathrm{FV}(q_i) \\ \{\overline{p}\ ;\ \overline{\lambda x.q}\} & \longrightarrow & q_i & \text{if } x_i \notin \mathrm{FV}(q_i) \end{array}$$

5 Extending the Curry-Howard isomorphism to definable rules

The optimizations for the logical rules, as given in Lemmas 9 and 12 can be extended to the proof terms and also to convertibilities and conversions. This gives us the possibility to capture questions related to normalization by looking at normalization for terms in the original calculus $\lambda^{\mathcal{C}}$. We will now describe the terms for the optimized rules in detail.

▶ **Definition 36.** For each optimization step in Lemmas 9 and 12 we give the canonical term for the optimized rule and its translation in terms of $\lambda^{\mathcal{C}}$ of Definition 27.

We first treat the two optimizations arising from Lemma 9, and then the optimization arising from Lemma 12.

- Given two rules

$$\frac{\vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n \qquad x_1 : B_1 \vdash q_1 : \Phi \ \ldots \ x_m : B_m \vdash q_m : \Phi \qquad z : A \vdash s : \Phi}{\vdash \{\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.s}\}_r : \Phi}\ \mathrm{in}_r$$

$$\frac{\vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n \qquad \vdash a : A \qquad x_1 : B_1 \vdash q_1 : \Phi \ \ldots \ x_m : B_m \vdash q_m : \Phi}{\vdash \{\overline{p, a}\ ;\ \overline{\lambda x.q}\}_{r'} : \Phi}\ \mathrm{in}_{r'}$$

  we have the following term for the optimized introduction rule

$$\frac{\vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n \qquad x_1 : B_1 \vdash q_1 : \Phi \ \ldots \ x_m : B_m \vdash q_m : \Phi}{\vdash \{\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.\{\overline{p, z}\ ;\ \overline{\lambda x.q}\}_{r'}}\}_r : \Phi}\ \mathrm{in}^{\mathrm{opt}}_{r,r'}$$

  We define the term $\{\overline{p}\ ;\ \overline{\lambda x.q}\}^{\circ}_{r,r'}$ as $\{\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.\{\overline{p, z}\ ;\ \overline{\lambda x.q}\}_{r'}}\}_r$
- Given two rules

$$\frac{\vdash t : \Phi \quad \vdash p_1 : A_1 \ldots \vdash p_n : A_n \quad x_1 : B_1 \vdash q_1 : D \ldots x_m : B_m \vdash q_m : D \quad z : A \vdash s : D}{\vdash t \cdot_r [\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.s}] : D}\ \mathrm{el}_r$$

$$\frac{\vdash t : \Phi \quad \vdash p_1 : A_1 \ldots \vdash p_n : A_n \quad \vdash a : A \quad x_1 : B_1 \vdash q_1 : D \ldots x_m : B_m \vdash q_m : D}{\vdash t \cdot_{r'} [\overline{p, a}\ ;\ \overline{\lambda x.q}] : D}\ \mathrm{el}_{r'}$$

  we have the following term for the optimized elimination rule

$$\frac{\vdash t : \Phi \qquad \vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n \qquad x_1 : B_1 \vdash q_1 : D \ \ldots \ x_m : B_m \vdash q_m : D}{\vdash t \cdot_r [\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.t \cdot_{r'} [\overline{p, z}\ ;\ \overline{\lambda x.q}]}] : D}\ \mathrm{el}^{\mathrm{opt}}_{r,r'}$$

  We define term $t \odot_{r,r'} [\overline{p}\ ;\ \overline{\lambda x.q}]$ as $t \cdot_r [\overline{p}\ ;\ \overline{\lambda x.q, \lambda z.t \cdot_{r'} [\overline{p, z}\ ;\ \overline{\lambda x.q}]}]$

- Given the rule

$$\frac{\vdash t : \Phi \qquad \vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n \qquad z : A \vdash s : D}{\vdash t \cdot_r [\overline{p} \,;\, \lambda z.s] : D}\,\mathrm{el}_r$$

we have the following term for the optimized elimination rule

$$\frac{\vdash t : \Phi \qquad \vdash p_1 : A_1 \ \ldots \ \vdash p_n : A_n}{\vdash t \cdot_r [\overline{p} \,;\, \lambda z.z] : A}\,\mathrm{el}_r^{\mathrm{opt}}$$

We define the term $t \boxdot_r [\overline{p}]$ as $t \cdot_r [\overline{p} \,;\, \lambda z.z]$

There is a canonical way in which the notions of detour convertibility and detour conversion extend to the optimized rules: the same rules as in Definition 29 apply. In case of a term of the form $\{\ldots \,;\, \ldots\} \cdot [\ldots \,;\, \ldots]$, a reduction is always possible, also in the case of optimized rules. For the permutation convertibilities, the situation is similar: the same rules as in Definition 32 apply.

▶ **Definition 37.** We define reduction on the optimized terms as follows. Let $\oslash$ be any $\cdot_{r''}$ or $\odot_{r'',r'''}$ for some r'', r'''. (For the notation, we refer to Definition 29.)

$$\begin{array}{rcl}
\text{For the } \ell = j \text{ case:} & & \\
\{\overline{p, p_j} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \oslash [\overline{s} \,;\, \overline{\lambda y.u, \lambda y_\ell.u_\ell}] & \longrightarrow_a & u_\ell[y_\ell := p_j] \\
\text{For the } k = i \text{ case:} & & \\
\{\overline{p} \,;\, \overline{\lambda x.q, \lambda x_i.q_i}\}^{\circ}_{r,r'} \oslash [\overline{s, s_k} \,;\, \overline{\lambda y.u}] & \longrightarrow_a & q_i[x_i := s_k] \oslash [\overline{s, s_k} \,;\, \overline{\lambda y.u}] \\
\text{For the } k = i \text{ case:} & & \\
\{\overline{s} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \boxdot_r [\overline{p}] & \longrightarrow_a & q_i[x_i := p_k] \boxdot_r [\overline{p}] \\
\text{Special case:} & & \\
\{\overline{s, s_j} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \boxdot_r [\overline{p}] & \longrightarrow_a & s_j
\end{array}$$

The last special case is when $\{\overline{s, s_j} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \boxdot_r [\overline{p}] : A$ and $s_j : A$. See the definition of $\{\overline{s, s_j} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \boxdot_r [\overline{p}]$ as $\{\overline{s, s_j} \,;\, \overline{\lambda x.q}\}^{\circ}_{r,r'} \cdot_r [\overline{p} \,;\, \lambda z.z]$ in Definition 36; this is the case where s_j matches the "invisible" $\lambda z.z$.

We also extend the notions of permutation convertibility and permutation conversion from Definition 25 (see also Definition 32): we add reduction rules for the optimized terms as follows.

$$(t \ominus [\overline{p} \,;\, \overline{\lambda x.q}]) \oslash [\overline{s} \,;\, \overline{\lambda y.u}] \quad \longrightarrow_b \quad t \ominus [\overline{p} \,;\, \overline{\lambda x.(q \oslash [\overline{s} \,;\, \overline{\lambda y.u}])}]$$

where $\ominus$ is any $\cdot_{r''}$ or $\odot_{r'',r'''}$ and $\oslash$ is any $\cdot_{r''}$ or $\odot_{r'',r'''}$ or $\boxdot_{r''}$.

▶ Remark. To clarify, we want to note explicitly that $t \boxdot_r [\overline{p}] \cdot_{r'} [\overline{q} \,;\, \overline{\lambda x.s}]$ does *not reduce to* $t \boxdot_r [\overline{p}]$. In case we only have the optimized rules, it does not reduce at all. If we consider $t \boxdot_r [\overline{p}]$ as a definition in the original calculus $\lambda^{\mathcal{C}}$, we do have a reduction,

$$t \boxdot_r [\overline{p}] \cdot_{r'} [\overline{q} \,;\, \overline{\lambda x.s}] \longrightarrow_b t \cdot_r [\overline{p} \,;\, \lambda z.z \cdot_{r'} [\overline{q} \,;\, \overline{\lambda x.s}]]$$

but this uses a non-optimized elimination.

▶ Remark. The process described in Definition 36, which is based on Lemmas 9 and 12 can be iterated, as we have seen in earlier examples. A simple way to view the rules for an n-ary connective c as a pair (b, r) where b is 0 or 1 and r is a partial function $r : \{1, 2, \ldots, n\} \to \{0, 1\}$. For a standard rule, derived from a line in the the truth table of c,

r is a total function. (If $r(i) = 1$, then A_i is a lemma in the rule and if $r(j) = 0$, then A_j is a case; if $b = 0$, we have an elimination rule, if $b = 1$ we have an introduction rule .) An optimized rule is a function r that is undefined for some elements of $\{1, \ldots, n\}$.

For the first case of Definition 36, where $\{\ldots\ ;\ \ldots\}^{\circ}_{r,r'}$ is defined in terms of $\{\ldots\ ;\ \ldots\}_r$ and $\{\ldots\ ;\ \ldots\}_{r'}$, we have $r'' = r \cap r'$ for the optimized rule r''. This is allowed in case $b = 1$ for r and r' and r and r' differ for only one element.

For the second case of Definition 36, where $\ldots \odot_{r,r'} [\ldots\ ;\ \ldots]$ is defined in terms of $\ldots \cdot_r [\ldots\ ;\ \ldots]$ and $\ldots \cdot_{r'} [\ldots\ ;\ \ldots]$, we again have $r'' = r \cap r'$ for the optimized rule r''. This is allowed in case $b = 0$ for r and r' and r and r' differ for only one element.

Optimization according to Lemma 12, the third case of Definition 36, corresponds with a (possibly partial) function r where $b = 0$ and $r(i) = 1$ for exactly one i.

With the definable optimized terms for elimination and introduction, we have a choice of taking these as defined terms, or taking them as primitives and removing the originals. Or even there is a third alternative of adding them as additional term constructions. After we have done some examples, we will, in Lemma 43, analyze the reduction behaviour of the newly defined terms in terms of the original ones.

Before that we state what the normal forms are of the optimized terms and the optimized reduction, extending Lemma 35. So in the following Lemma, we consider the situation where we have added optimized terms and reductions, while removing the original ones. The proof is straightforward, keeping in mind Remark 5 and the fact that with optimized terms, if an introduction is followed immediately by an elimination, then there is a "matching case" that allows us to reduce the term.

▶ **Lemma 38.** *We simultaneously characterize* $\mathsf{NF}^{\mathrm{opt}}$*, the set of* terms in normal form *of* $\mathsf{IPC}_{\mathcal{C}}$ *with optimized terms and reductions, and the set of* neutral terms *inductively as follows.*

- $x \in \mathsf{NF}^{\mathrm{opt}}$ *and x is neutral, for every variable x,*
- $\{\overline{p}\ ;\ \overline{\lambda y.q}\} \in \mathsf{NF}^{\mathrm{opt}}$ *if all p_i and q_j are in* $\mathsf{NF}^{\mathrm{opt}}$*,*
- $x \oslash [\overline{p}\ ;\ \overline{\lambda y.q}] \in \mathsf{NF}^{\mathrm{opt}}$ *if all p_i and q_j are in* $\mathsf{NF}^{\mathrm{opt}}$ *and x is a variable; this term is neutral if $\oslash = \boxdot_r$ for some r.*
- $t \boxdot_r [\overline{s}] \oslash [\overline{p}\ ;\ \overline{\lambda y.q}] \in \mathsf{NF}^{\mathrm{opt}}$ *if all s_k, p_i and q_j are in* $\mathsf{NF}^{\mathrm{opt}}$ *and t is neutral; this term is neutral if $\oslash = \boxdot_{r'}$ for some r'.*

What the Lemma says is that terms like

$$x \boxdot_r [\overline{s_1}] \boxdot_{r'} [\overline{s_2}] \boxdot_{r''} \ldots\ldots \oslash [\overline{p}\ ;\ \overline{\lambda y.q}]$$

are also normal forms, if $\overline{s_1}$, $\overline{s_2}, \ldots, \overline{p}$ and $\overline{q}$ are.

▶ **Example 39.** We continue Example 30 and look into the optimized rules for $\wedge$, as given in Definition 13. The introduction rule of Example 30 is the same as in Definition 13; the usual "pairing" construction is given by $\{a, b\ ;\ \}^{\wedge}$. For elimination, we would like to have the following "projection" rules.

$$\frac{\vdash t : A \wedge B}{\vdash \pi_1\, t : A} \qquad \frac{\vdash t : A \wedge B}{\vdash \pi_2\, t : B}$$

That is, we would like to define $\pi_1\, t$ and $\pi_2\, t$ in terms of the constructions of Example 30, with the expected reduction rules: $\pi_1\, \{a, b\ ;\ \}^{\wedge} \longrightarrow_a a$ and $\pi_2\, \{a, b\ ;\ \}^{\wedge} \longrightarrow_a b$. Definition 36 gives the clue. Let's consider the first projection, $\pi_1\, t$. We have the following optimization of the $\wedge$-rules of Example 30.

$$\frac{\vdash t : A \wedge B \qquad x : A \vdash p : D}{t \odot^{\wedge}_{a} [\ ;\ \lambda x^{A}.p] : D}$$

where $t \odot_a^\wedge [\ ;\ \lambda x^A.p] := t \cdot_1^\wedge [\ ;\ \lambda x^A.p, \lambda z^B.t \cdot_3^\wedge [z\ ;\ \lambda x^A.p]]$. It is easily verified that we have the following reduction

$$\{a, b\ ;\ \}^\wedge \odot_a^\wedge [\ ;\ \lambda x^A.p] \longrightarrow_a p[x := a].$$

We have another optimization:

$$\frac{\vdash t : A \wedge B}{\vdash t \boxdot_1^\wedge [\ ;\ ] : A}$$

where $t \boxdot_1^\wedge [\ ;\ ] := t \odot_a^\wedge [\ ;\ \lambda x^A.x]$.

All together we have

$$\pi_1\, t := t \boxdot_1^\wedge [\ ;\ ] = t \odot_a^\wedge [\ ;\ \lambda x^A.x] = t \cdot_1^\wedge [\ ;\ \lambda x^A.x, \lambda z^B.t \cdot_3^\wedge [z\ ;\ \lambda x^A.x]]$$

which has the following reductions.

$$\begin{array}{lcl}
\pi_1\, \{a, b\ ;\ \}^\wedge & = & \{a, b\ ;\ \}^\wedge \cdot_1^\wedge [\ ;\ \lambda x^A.x, \lambda z^B.\{a, b\ ;\ \}^\wedge \cdot_3^\wedge [z\ ;\ \lambda x^A.x]] \\
 & \longrightarrow_a & a \\
\pi_1\, \{a, b\ ;\ \}^\wedge & = & \{a, b\ ;\ \}^\wedge \cdot_1^\wedge [\ ;\ \lambda x^A.x, \lambda z^B.\{a, b\ ;\ \}^\wedge \cdot_3^\wedge [z\ ;\ \lambda x^A.x]] \\
 & \longrightarrow_a & \{a, b\ ;\ \}^\wedge \cdot_3^\wedge [b\ ;\ \lambda x^A.x] \\
 & \longrightarrow_a & a
\end{array}$$

Similarly, we define $\pi_2\, t := t \cdot_1^\wedge [\ ;\ \lambda x^B.x, \lambda z^A.t \cdot_2^\wedge [z\ ;\ \lambda x^B.x]]$. Then $\pi_2\, \{a, b\ ;\ \}^\wedge \longrightarrow_a^+ b$.

An interesting feature is that the reduction rules for our non-optimized calculus are not Church-Rosser, as we have already indicated in Example 30 and also in Example 23. On the other hand, the optimized rules for standard intuitionistic proposition logic are know to be Church-Rosser. We look into the case for $\wedge$ in more detail.

▶ **Example 40.** The set of full rules for $\wedge$, see Example 30, is not Church-Rosser as the following concrete example shows. Suppose we have $\vdash p : D$ and $\vdash q : D$, where p and q are different.

$$\frac{\dfrac{a : A \vdash a : A \qquad b : B \vdash b : B}{a : A, b : B \vdash \{a, b\ ;\ \}^\wedge : A \wedge B} \qquad x : A \vdash p : D \qquad y : B \vdash q : D}{\{a, b\ ;\ \}^\wedge \cdot_1^\wedge [\ ;\ \lambda x^A.p, \lambda y^B, q]}$$

This term reduces to both p and q, which are distinct terms of type D. The crucial point is in the rule for $- \cdot_1^\wedge [\ ;\ -]$ that admits a choice:

$$\frac{\vdash t : A \wedge B \quad x : A \vdash p : D \quad y : B \vdash q : D}{\vdash t \cdot_1^\wedge [\ ;\ \lambda x.p, \lambda y.q] : D}$$

For $t = \{a, b\ ;\ \}^\wedge$ we can either select the "A-case" or the "B-case".

We have shown how the optimized rules can be explained in terms of the full rules, but we can also doe the opposite: interpret the full rules for $\wedge$ of Example 30 in terms of π_1 and π_2. Then we get

$$\begin{array}{rcl}
t \cdot_1^\wedge [\ ;\ \lambda x^A.p, \lambda y^B.q] & := & p[x := \pi_1\, t] \\
t \cdot_2^\wedge [a'\ ;\ \lambda y^B.q] & := & q[y := \pi_2\, t] \\
t \cdot_3^\wedge [b'\ ;\ \lambda x^A.p] & := & p[x := \pi_1\, t]
\end{array}$$

where in the first case we could also have chosen $q[y := \pi_2\, t]$. We observe that the non-determinism in the full rules is resolved by a choice we make in the translation of the first $\wedge$-elimination.

▶ **Example 41.** We now look into the optimized rules for implication of Definition 13. The full rules have been treated in Example 31. We want to define the following terms.

$$\frac{x : A \vdash p : A \to B}{\vdash \{\,;\, \lambda x^A.p\}_1^{\to\circ} : A \to B} \qquad \frac{\vdash b : B}{\vdash \{b \,;\,\}_2^{\to\circ} : A \to B} \qquad \frac{\vdash t : A \to B \quad \vdash a : A}{\vdash t \boxdot^{\to} [a] : B}$$

These can be defined from the terms in Example 31 via the optimizations of Definition 36 as follows.

$$\begin{aligned}
\{\,;\, \lambda x^A.p\}_1^{\to\circ} &:= \{\,;\, \lambda x^A.p, \lambda z.\{z \,;\, \lambda x^A.p\}_2^{\to}\}_1^{\to}\\
\{b \,;\,\}_2^{\to\circ} &:= \{b \,;\, \lambda z^A.\{z, b \,;\,\}_3^{\to}\}_2^{\to}\\
t \boxdot^{\to} [a] &:= t \cdot^{\to} [a \,;\, \lambda z.z]
\end{aligned}$$

These obey the following reductions.

$$\begin{aligned}
\{\,;\, \lambda x^A.p\}_1^{\to\circ} \boxdot^{\to} [a] &= \{\,;\, \lambda x^A.p, \lambda z.\{z \,;\, \lambda x^A.p\}_2^{\to}\}_1^{\to} \cdot^{\to} [a \,;\, \lambda z.z]\\
&\longrightarrow_a p[x := a] \cdot^{\to} [a \,;\, \lambda z.z]\\
&= p[x := a] \boxdot^{\to} [a]\\
\{b \,;\,\}_2^{\to\circ} \boxdot^{\to} [a] &:= \{b \,;\, \lambda z^A.\{z, b \,;\,\}_3^{\to}\}_2^{\to} \boxdot^{\to} [a]\\
&\longrightarrow_a b\\
\{b \,;\,\}_2^{\to\circ} \boxdot^{\to} [a] &:= \{b \,;\, \lambda z^A.\{z, b \,;\,\}_3^{\to}\}_2^{\to} \boxdot^{\to} [a]\\
&\longrightarrow_a \{a, b \,;\,\}_3^{\to} \boxdot^{\to} [a]\\
&\longrightarrow_a b
\end{aligned}$$

These are the exact reduction rules one would expect for these terms. We can again translate these to the well-known β-rules, that we will define in Definition 47.

The definition of the standard rule for $\to$-introduction essentially uses the $\boxdot$ construction, which has a somewhat special behaviour under normalization, as we have seen in Remark 5 and Lemma 38. Let's look at an example to emphasis this.

▶ **Example 42.** Consider the following proof.

$$\frac{\dfrac{t : A \to B \to C \qquad \vdash a : A}{t \boxdot^{\to} [a] : B \to C} \qquad \vdash b : B}{t \boxdot^{\to} [a] \boxdot^{\to} [b] : C}$$

If t is not an introduction term ($t \neq \{\lambda x.q\}^{\to}$), then this is not a redex with the optimized rules. However, in case $\boxdot$ is a defined term-construction, this term is reducible:

$$t \boxdot^{\to} [a] \boxdot^{\to} [b] \longrightarrow_b t \cdot^{\to} [a \,;\, \lambda z^{B\to C}.z \boxdot^{\to} [b]].$$

To clarify, the derivation for this term is:

$$\frac{\vdash t : A \to B \to C \qquad \vdash a : A \qquad \dfrac{z : B \to C \vdash z : B \to C \qquad \vdash b : B}{z : B \to C \vdash z \boxdot^{\to} [b] : C}}{t \cdot^{\to} [a \,;\, \lambda z^{B\to C}.z \boxdot^{\to} [b]] : C}$$

▶ **Lemma 43.** *The translation of an $\longrightarrow_a$ step in the optimized calculus translates to a (possibly multistep) $\longrightarrow_a$ step in the original calculus $\lambda^{\mathcal{C}}$.*

Proof. We show two cases:

1. If $\{\bar{t} \,;\, \overline{\lambda y.v}\}^{\circ}_{r_3,r_4} \odot_{r_1,r_2} [\overline{p} \,;\, \overline{\lambda x.q}] \longrightarrow_a R$ (using the optimized rules) and $\{\bar{t} \,;\, \overline{\lambda y.r}\}^{\circ}_{r_3,r_4} \odot_{r_1,r_2} [\overline{p} \,;\, \overline{\lambda x.q}]$ translates to T in the original calculus $\lambda^{\mathcal{C}}$, then there is a term T' such that $T \longrightarrow_a^+ T'$ and R translates to T' in $\lambda^{\mathcal{C}}$. Here $\longrightarrow_a^+$ denotes a non-zero sequence of reductions.

   In this case the translation T is as follows. $T = M \cdot [\overline{p} \,;\, \overline{\lambda x.q, \lambda z.\overline{M \cdot [\overline{p,z} \,;\, \overline{\lambda x.q}]}}]$, where we abbreviate $M := \{\bar{t} \,;\, \overline{\lambda y.v, \lambda z.\{\overline{t,z} \,;\, \overline{\lambda y.v}\}}\}$. There are two possible cases for the reduction.
   - Case $\{\bar{t} \,;\, \overline{\lambda y.v}\}^{\circ}_{r_3,r_4} \odot_{r_1,r_2} [\overline{p} \,;\, \overline{\lambda x.q}] \longrightarrow_a q_\ell[x_\ell := t_j]$. Then $T \longrightarrow_a q_\ell[x_\ell := t_j]$ and we are done.
   - Case $\{\bar{t} \,;\, \overline{\lambda y.v}\}^{\circ}_{r_3,r_4} \odot_{r_1,r_2} [\overline{p} \,;\, \overline{\lambda x.q}] \longrightarrow_a v_i[y_i := p_k] \odot_{r_1,r_2} [\overline{p} \,;\, \overline{\lambda x.q}]$. Then
     $$\begin{aligned} T &\longrightarrow_a v_i[y_i := p_k] \cdot [\overline{p} \,;\, \overline{\lambda x.q, \lambda z.M \cdot [\overline{p,z} \,;\, \overline{\lambda x.q}]}] \\ &\longrightarrow_a v_i[y_i := p_k] \cdot [\overline{p} \,;\, \overline{\lambda x.q, \lambda z.v_i[y_i := p_k] \cdot [\overline{p,z} \,;\, \overline{\lambda x.q}]}] \end{aligned}$$
     and we are done.
2. If $\{\bar{t} \,;\, \overline{\lambda y.v}\}^{\circ}_{r_2,r_3} \odot_{r_1} [\overline{p}] \longrightarrow_a R$ and $\{\bar{t} \,;\, \overline{\lambda y.r}\}^{\circ}_{r_2,r_3} \odot_{r_1} [\overline{p}]$ translates to T in the original calculus $\lambda^{\mathcal{C}}$, then there is a term T' such that $T \longrightarrow_a^+ T'$ and R translates to T' in $\lambda^{\mathcal{C}}$.
   Now the translation T is as follows. $T = \{\bar{t} \,;\, \overline{\lambda y.v, \lambda z.\{\overline{t,z} \,;\, \overline{\lambda y.v}\}}\} \cdot [\overline{p} \,;\, \lambda z.z]$. There is one possibility for the reduction.
   - Case $\{\bar{t} \,;\, \overline{\lambda y.v}\}^{\circ}_{r_2,r_3} \odot_{r_1} [\overline{p}] \longrightarrow_a v_i[y_i := p_k] \odot_{r_1} [\overline{p}]$. Then
     $$T \longrightarrow_a v_i[y_i := p_k] \cdot [\overline{p} \,;\, \lambda z.z]$$
     and we are done. ◀

As mentioned, Schroeder-Heister[17] has proposed another elimination rule for $\wedge$ which is slightly different from ours. Von Plato [23] calls this *general elimination* while Tennant [21] calls it *parallel elimination.* We call it parallel $\wedge$-elimination and give it in typed λ-calculus format.

▶ **Definition 44.** We define the *parallel* $\wedge$*-elimination rule* as follows

$$\frac{\Gamma \vdash t : A \wedge B \qquad \Gamma, x : A, y : B \vdash q : D}{\Gamma \vdash t \cdot^{\text{par}} [\lambda x, y.q] : D}\ \wedge\text{-el}$$

The reduction (detour conversion) rule associated with this rule is as follows.

$$\{a, b \,;\, \} \cdot^{\text{par}} [\lambda x, y.q] \longrightarrow_{\text{par}} q[x := a, y := b].$$

We show that this elimination rule can be translated in terms of ours and that reduction is preserved.

▶ **Definition 45.** We translate the parallel $\wedge$-elimination rule of Definition 44 by defining it in terms of the optimized terms for $\wedge$ of Example 39. We consider the following optimized rules, the first of which was given explicitly in Example 39.

$$\frac{\Gamma \vdash t : A \wedge B \qquad \Gamma, x : A \vdash q : D}{\Gamma \vdash t \odot^{\wedge}_{a} [\,;\, \lambda x.q] : D} \qquad \frac{\Gamma \vdash t : A \wedge B \qquad \Gamma, y : B \vdash q : D}{\Gamma \vdash t \odot^{\wedge}_{b} [\,;\, \lambda y.q] : D}$$

Now define

$$t \cdot^{\text{par}} [\lambda x, y.q] := t \odot^{\wedge}_{a} [\,;\, \lambda x.\ t \odot^{\wedge}_{b} [\,;\, \lambda y.q]].$$

▶ **Lemma 46.** *The defined term* $t \cdot^{\text{par}} [\lambda x, y.q]$ *is of the right type and the translation of an* $\longrightarrow_{\text{par}}$ *step in the calculus with the parallel* $\wedge$*-elimination rule translates to multistep* $\longrightarrow_a$ *in the original calculus* $\lambda^{\mathcal{C}}$*.*

Proof. Given $\Gamma \vdash t : A \wedge B$ and $\Gamma, x : A, y : B \vdash q : D$, we have

$$\dfrac{\Gamma \vdash t : A \wedge B \qquad \dfrac{\Gamma, x : A \vdash t : A \wedge B \qquad \Gamma, x : A, y : B \vdash q : D}{\Gamma, x : A \vdash t \odot_b^\wedge [\,;\, \lambda y.q] : D}}{\Gamma \vdash t \odot_a^\wedge [\,;\, \lambda x.\, t \odot_b^\wedge [\,;\, \lambda y.q]] : D}$$

The reduction can easily be verified:

$$\begin{array}{rcl} \{a, b\,;\,\}^\wedge \cdot^{\mathrm{par}} [\lambda x, y.q] & := & \{a, b\,;\,\}^\wedge \odot_a^\wedge [\,;\, \lambda x.\, \{a, b\,;\,\}^\wedge \odot_b^\wedge [\,;\, \lambda y.q]] \\ & \longrightarrow_a & \{a, b\,;\,\}^\wedge \odot_b^\wedge [\,;\, \lambda y.q[x := a]] \\ & \longrightarrow_a & q[x := a, y := b]. \end{array}$$ ◀

We define the standard rule for →-introduction and show that this introduction rule can be translated in terms of ours and that the reduction is preserved.

▶ **Definition 47.** We define the *standard rule for →-introduction* as follows, where we describe it using terms.

$$\dfrac{\Gamma, x : A \vdash q : B}{\Gamma \vdash \{\lambda x.q\}^\to : A \to B}\ \to\text{-in}$$

The reduction rule associated with this term is as follows.

$$\{\lambda x.q\}^\to \boxdot^\to [a] \longrightarrow_s q[x := a],$$

where $t \boxdot^\to [a]$ is the optimized elimination rule from Example 41.

▶ **Definition 48.** We define the standard →-introduction rule in terms of optimized →-rules (Example 41) as follows. Given $\Gamma, x : A \vdash q : B$ we define

$$\{\lambda x.q\}^\to := \{\,;\, \lambda x.\{q\,;\,\}_2^{\to\circ}\}_1^{\to\circ}.$$

▶ **Lemma 49.** *The translation of $\{\lambda x.q\}^\to$ is well-typed and the translation of an $\longrightarrow_s$ step in the calculus with the standard rule for → translates to multistep $\longrightarrow_a$ in the original calculus λ^{C}.*

Proof. The well-typedness is easily verified:

$$\dfrac{\dfrac{x : A \vdash q : B}{x : A \vdash \{q\,;\,\}_2^{\to\circ} : A \to B}}{\vdash \{\,;\, \lambda x^A.\{q\,;\,\}_2^{\to\circ}\}_1^{\to\circ} : A \to B}$$

For the reduction:

$$\{\,;\, \lambda x^A.\{q\,;\,\}_2^{\to\circ}\}_1^{\to\circ} \cdot^\to [a\,;\,] \longrightarrow_a \{q[x := a]\,;\,\}_2^{\to\circ} \cdot^\to [a\,;\,] \longrightarrow_a q[x := a].$$ ◀

We define the traditional rule for ¬-introduction and show that it can be translated in terms of ours and that detour conversion is preserved.

▶ **Definition 50.** We define the *traditional rules for ¬*, the introduction and the elimination rule, as follows, where we describe them using terms.

$$\dfrac{\Gamma, x : A \vdash t : \neg B \qquad \Gamma, y : A \vdash q : B}{\Gamma \vdash \{\lambda x.t, \lambda y.q\}^{\mathrm{t}} : \neg A} \qquad \dfrac{\Gamma \vdash t : \neg A \qquad \Gamma \vdash a : A}{\Gamma \vdash t \cdot^\neg [a\,;\,] : D}$$

The reduction rule associated with these terms is as follows.

$$\{\lambda x^A.t, \lambda y^A.q\}^{\mathrm{t}} \cdot^\neg [a\,;\,] \longrightarrow_\neg t[x := a] \cdot^\neg [q[y := a]\,;\,].$$

▶ **Example 51.** The rules for negation that we derive from our general Definition 27 are the following.

$$\frac{\Gamma, x : A \vdash q : \neg A}{\Gamma \vdash \{\ ;\ \lambda x.q\}^{\neg} : \neg A} \qquad \frac{\Gamma \vdash t : \neg A \qquad \Gamma \vdash a : A}{\Gamma \vdash t \cdot^{\neg} [a\ ;\ ] : D}$$

with reduction

$$\{\ ;\ \lambda x^A.q\}^{\neg} \cdot^{\neg} [a\ ;\ ] \longrightarrow_a q[x := a] \cdot^{\neg} [a\ ;\ ].$$

We see that the elimination rule for $\neg$ in Example 51 is the same as the traditional one. The traditional introduction rule for $\neg$ is definable.

▶ **Definition 52.** We define the traditional $\neg$-introduction rule in terms of the one of Example 51 as follows. Given $\Gamma, x : A \vdash t : \neg B$ and $\Gamma, y : A \vdash q : B$ we define

$$\{\lambda x^A.t, \lambda y^A.q\}^{\mathrm{t}} := \{\ ;\ \lambda x^A.t \cdot^{\neg} [q[y := x]\ ;\ ]\}^{\neg}$$

▶ **Lemma 53.** *The definition of $\{\lambda x.t, \lambda y.q\}^{\mathrm{t}}$ is well-typed and a $\longrightarrow_{\neg}$ step in the calculus with the traditional rule for $\neg$ translates to multistep $\longrightarrow_a$ in the original calculus λ^{C}.*

Proof. For the well-typedness:

$$\dfrac{\dfrac{\Gamma, x : A \vdash t : \neg B \qquad \dfrac{\Gamma, y : A \vdash q : B}{\Gamma, x : A \vdash q[y := x] : B}}{\Gamma, x : A \vdash t \cdot^{\neg} [q[y := x]\ ;\ ] : \neg A}\ \neg\text{-el}}{\Gamma \vdash \{\ ;\ \lambda x^A.t \cdot^{\neg} [q[y := x]\ ;\ ]\}^{\neg} : \neg A}\ \neg\text{-in}$$

For the reduction:

$$\{\ ;\ \lambda x^A.t \cdot^{\neg} [q[y := x]\ ;\ ]\}^{\neg} \cdot^{\neg} [a\ ;\ ] \longrightarrow_a t[x := a] \cdot^{\neg} [q[x := a]\ ;\ ].$$ ◀

As a final example, we give the proof-terms for the optimized rules of nand-logic, as described in Definition 14.

▶ **Example 54.** The proof-terms for nand-logic are

$$\frac{x : A \vdash p : A \uparrow B}{\vdash \{\ ;\ \lambda x^A.p\}^{\uparrow} : A \uparrow B} \qquad \frac{y : B \vdash q : A \uparrow B}{\vdash \{\ ;\ \lambda y^B.q\}^{\uparrow} : A \uparrow B} \qquad \frac{\vdash t : A \uparrow B \quad \vdash a : A \quad \vdash b : B}{\vdash t \cdot^{\uparrow} [a, b\ ;\ ] : D}$$

with reduction rules

$$\begin{aligned} \{\ ;\ \lambda x^A.p\}^{\uparrow} \cdot^{\uparrow} [a, b\ ;\ ] &\longrightarrow_a p[x := a] \cdot^{\uparrow} [a, b\ ;\ ] \\ \{\ ;\ \lambda y^B.q\}^{\uparrow} \cdot^{\uparrow} [a, b\ ;\ ] &\longrightarrow_a q[y := b] \cdot^{\uparrow} [a, b\ ;\ ] \end{aligned}$$

This time we have a situation where a permutation conversion actually reduces the size of a term considerably. Suppose $t : A \uparrow B$ and $a : A$, $b : B$, $c : C$, $d : D$. Then we have

$$\dfrac{\dfrac{\vdash t : A \uparrow B \quad \vdash a : A \quad \vdash b : B}{\vdash t \cdot^{\uparrow} [a, b\ ;\ ] : C \uparrow D} \qquad \vdash c : C \qquad \vdash d : D}{t \cdot^{\uparrow} [a, b\ ;\ ] \cdot^{\uparrow} [c, d\ ;\ ] : E}$$

We have

$$t \cdot^{\uparrow} [a, b\ ;\ ] \cdot^{\uparrow} [c, d\ ;\ ] \longrightarrow_b t \cdot^{\uparrow} [a, b\ ;\ ]$$

which is of type E, and we see that the superfluous second nand-elimination rule has been removed.

As another example, we can give a proof-term of $A \vee \neg A^{\uparrow}$, the proposition in nand-logic that we have shown to be provable after the proof of Proposition 17. It's proof-term is

$$\{\ ;\ \lambda x.\{\ ;\ \lambda y.y \cdot^{\uparrow} [x, x\ ;\ ]\}^{\uparrow}\}^{\uparrow} : (A \uparrow A) \uparrow (\dot{\neg} A \uparrow \dot{\neg} A)$$

6 Normalization

In this section we prove that $\longrightarrow_a$ and $\longrightarrow_b$ are both strongly normalizing (SN). We also give a proof of weak normalization (WN) of the combination of $\longrightarrow_a$ and $\longrightarrow_b$. As usual, SN states that there are no terms that have an infinite reduction path, and WN states that for each term there is a reduction path that leads to a normal form. For the proof of WN we describe an actual procedure for finding a normal form of a term.

▶ **Theorem 55.** *The reduction $\longrightarrow_b$ is strongly normalizing.*

Proof. We define a measure $|-|$ from terms to natural numbers that decreases with every reduction step. For notational convenience we suppress the reference to the derivation rule r.

$$\begin{array}{rcl} |x| & := & 1 \\ |\{\overline{p}\ ;\ \overline{\lambda y.q}\}| & := & \Sigma|p_i| + \Sigma|q_j| \\ |t \cdot [\overline{s}\ ;\ \overline{\lambda y.u}]| & := & |t|(2 + \Sigma|s_k| + \Sigma|u_\ell|) \end{array}$$

It can easy be verified that, if $t_0 \longrightarrow_b t_1$, then $|t_0| > |t_1|$, so $\longrightarrow_b$ is strongly normalizing. ◀

▶ **Corollary 56.** *The reduction $\longrightarrow_b$ for the optimized rules of Definition 36, the standard rule for $\to$-elimination of Definition 47, the parallel $\wedge$-elimination rule of Definition 44 and the traditional rule for $\neg$-elimination of Definition 50 are strongly normalizing.*

Proof. The same metrics as in the proof of Theorem 55 applies. For the parallel reduction, define $|t \cdot^{\text{par}} [\lambda x, y.q]| := |t|(2 + |q|)$. ◀

6.1 Strong Normalization of the detour conversion

We now prove strong normalization for $\longrightarrow_a$ by adapting the well-known *saturated sets method* of Tait [20] and Girard [8] to our calculus. Recall that Term is the set of all untyped proof-terms. (Definition 27.) We write SN for the set of strongly normalizing (untyped) terms and we write Var for the set of variables.

▶ **Definition 57.** **1.** The set Neut of *neutral terms* is defined by
  a. Var $\subseteq$ Neut,
  b. $t \cdot [\overline{p}\ ;\ \overline{\lambda y.q}] \in$ Neut for all $t \in$ Neut and $\overline{p}, \overline{\lambda y.q} \in$ SN.

2. The term t does a *key reduction* to t', notation $t \longrightarrow_a^k t'$, in case
  a. t is a redex itself (according to Definition 29) and t' is its reduct,
  b. $t = t_0 \cdot [\overline{p}\ ;\ \overline{\lambda y.q}]$, $t' = t_1 \cdot [\overline{p}\ ;\ \overline{\lambda y.q}]$ and $t_0 \longrightarrow_a^k t_1$.

3. A set $X \subseteq$ Term is *saturated* ($X \in$ SAT) if it satisfies the following properties
  a. $X \subseteq$ SN,
  b. Neut $\subseteq X$
  c. X is closed under *key-redex expansion*: if $t \in$ SN and $\forall q(t \longrightarrow_a^k q \Rightarrow q \in X)$, then $t \in X$.

4. For a connective c of arity n and $X_1, \ldots, X_n \in$ SAT we define the set $c(X_1, \ldots, X_n)$ as follows. Assume that $r_1, \ldots, r_m$ are the elimination rules for c.

$$\begin{array}{l} c(X_1, \ldots, X_n) := \{t \mid \forall r_i \in \{r_1, \ldots, r_m\} \\ \qquad\qquad \forall D \in \mathsf{SAT}, \forall \overline{p}, \overline{q} \in \mathsf{Term} \\ \forall k(p_k \in X_k) \wedge (\forall \ell \forall u_\ell \in X_\ell\, (q_\ell[y_\ell := u_\ell] \in D)) \quad \Longrightarrow \quad t \cdot_{r_i} [\overline{p}\ ;\ \overline{\lambda y.q}] \in D\ \} \end{array}$$

In the definition of $c(X_1,\ldots,X_n)$ it should be clear that we quantify over all elimination rules for the connective c. In the quantification $\forall\overline{p},\overline{q}\in\mathsf{Term}$ we could also quantify over $\forall\overline{p},\overline{q}\in\mathsf{SN}$: it amounts to the same because the additional conditions $\forall k(p_k\in X_k)$ and $\forall\ell\,\forall u_\ell\in X_\ell\,(q_\ell[y_\ell:=u_\ell]\in D$ imply that $\overline{p},\overline{q}\in\mathsf{SN}$.

▶ **Lemma 58.** *If $X_1,\ldots,X_n\in\mathsf{SAT}$, then $c(X_1,\ldots,X_n)\in\mathsf{SAT}$.*

Proof. We check the 3 conditions for $c(X_1,\ldots,X_n)$. Suppose $X_1,\ldots,X_n\in\mathsf{SAT}$.

a. That $c(X_1,\ldots,X_n)\subseteq\mathsf{SN}$ follows directly from the fact that if $t\in c(X_1,\ldots,X_n)$, then $t\cdot[\overline{p}\;;\;\overline{\lambda x.q}]\in D$ and $D\subseteq\mathsf{SN}$, so $t\cdot[\overline{p}\;;\;\overline{\lambda x.q}]\in\mathsf{SN}$, so $t\in\mathsf{SN}$.

b. For $t\in\mathsf{Neut}$ and $D\in\mathsf{SAT}$ and $\overline{p},\overline{q}\in\mathsf{SN}$ with $\forall k(p_k\in X_k)$ and $\forall\ell\,\forall u_\ell\in X_\ell\,(q_\ell[y_\ell:=u_\ell]\in D)$, we have $t\cdot_{r_i}[\overline{p}\;;\;\overline{\lambda y.q}]\in\mathsf{Neut}\subseteq D$, so we can conclude that $t\in c(X_1,\ldots,X_n)$.

c. Suppose $t\in\mathsf{SN}$ and $\forall t'(t\longrightarrow_a^k t'\Rightarrow t'\in c(X_1,\ldots,X_n))$ (*). Let r_i be a rule for c and let $D\in\mathsf{SAT}$, $\overline{p},\overline{q}\in\mathsf{Term}$ with $\forall k(p_k\in X_k)$ and $\forall\ell\,\forall u_\ell\in X_\ell\,(q_\ell[y_\ell:=u_\ell]\in D)$. For all t' with $t\longrightarrow_a^k t'$ we have $t\cdot_{r_i}[\overline{p}\;;\;\overline{\lambda y.q}]\longrightarrow_a^k t'\cdot_{r_i}[\overline{p}\;;\;\overline{\lambda y.q}]$ and $t'\cdot_{r_i}[\overline{p}\;;\;\overline{\lambda y.q}]\in D$ by (*). So, $t\cdot_{r_i}[\overline{p}\;;\;\overline{\lambda y.q}]\in D$ and so $t\in c(X_1,\ldots,X_n)$. ◀

We use the saturated sets as a semantics for types: if A is a type, $\langle A\rangle$ will be a saturated set. The simplest way to do this is to interpret all type variables (proposition letters) as the set SN, which is indeed a saturated set.

▶ **Definition 59.** For A a type, we define $\langle A\rangle$ by induction on A as follows.

- $\langle A\rangle:=\mathsf{SN}$ if A is a proposition letter.
- $c(A_1,\ldots,A_n):=c(\langle A_1\rangle,\ldots,\langle A_n\rangle)$, where the right hand side is the interpretation of the connective c on saturated sets, as given in Definition 57, case (4).

We will often confuse A and $\langle A\rangle$, to avoid notational overhead, and just identify the proposition A with its interpretation as a saturated set $\langle A\rangle$.

▶ **Definition 60.** Given a context Γ, a map (valuation) $\rho:\mathsf{Var}\to\mathsf{Term}$ satisfies Γ, notation $\rho\models\Gamma$, in case $\rho(x)\in\langle A\rangle$ for all $x:A\in\Gamma$.

If $t\in\mathsf{Term}$ and $\rho:\mathsf{Var}\to\mathsf{Term}$, we write $\langle t\rangle_\rho$ for t where ρ has been carried out as a substitution on t.

A valuation $\rho:\mathsf{Var}\to\mathsf{Term}$ is only relevant for a finite number of variables: those that are declared in the context Γ under consideration. So we will always assume that $\rho(x)\neq x$ only for a finite number of $x\in\mathsf{Var}$. Those x we call the *support of* ρ. When applying ρ as a substitution to a term t we may need to "go under a λ", e.g. when applying ρ to $\{\overline{p}\;;\;\overline{\lambda x.q}\}$ In this case we always assume that the bound variable is not in the support of ρ. (We can always rename it.)

▶ **Lemma 61.** *If $\Gamma\vdash t:A$, and $\rho\models\Gamma$, then $\langle t\rangle_\rho\in\langle A\rangle$.*

Proof. By induction on the derivation of $\Gamma\vdash t:A$. Suppose $\rho\models\Gamma$. For the (axiom) case, it is trivial. We ignore ρ for the rest of the proof, as it gives a lot of notational overhead, so we just write t for $\langle t\rangle_\rho$.

- Suppose $\Phi=c(A_1,\ldots,A_n)$ and

$$\frac{\ldots\Gamma\vdash s_j:A_j\ldots\qquad\ldots\Gamma,x_i:A_i\vdash t_i:\Phi\ldots}{\Gamma\vdash\{\overline{s}\;;\;\overline{\lambda x.t}\}_r:\Phi}\text{ in}$$

Let r' be a rule for c, $D\in\mathsf{SAT}$, $\overline{p},\overline{q}\in\mathsf{Term}$ with $\forall k(p_k\in A_k)$ and $\forall\ell\,\forall u_\ell\in A_\ell\,(q_\ell[y_\ell:=u_\ell]\in D)$. For $\{\overline{s}\;;\;\overline{\lambda x.t}\}_r\cdot_{r'}[\overline{p}\;;\;\overline{\lambda y.q}]$ there are the following possible key-reductions:

$$\{\overline{s}\;;\;\overline{\lambda x.t}\}_r\cdot_{r'}[\overline{p}\;;\;\overline{\lambda y.q}]\quad\longrightarrow_a^k\quad q_l[y_l:=s_j]\tag{1}$$

$$\{\overline{s}\;;\;\overline{\lambda x.t}\}_r\cdot_{r'}[\overline{p}\;;\;\overline{\lambda y.q}]\quad\longrightarrow_a^k\quad t_i[x_i:=p_k]\cdot_{r'}[\overline{p}\;;\;\overline{\lambda y.q}]\tag{2}$$

In case (1), $q_l[y_l := s_j] \in D$ by the assumption and the induction hypothesis. In case (2), $t_i[x_i := p_k] \in \Phi$ by the induction hypothesis and so $t_i[x_i := p_k] \cdot_{r'} [\overline{p} \,;\, \overline{\lambda y.q}] \in D$ by the definition of $\Phi = c(A_1, \ldots, A_n)$ as a saturated set. So, $\{\overline{s} \,;\, \overline{\lambda x.t}\}_r \cdot_{r'} [\overline{p} \,;\, \overline{\lambda y.q}] \in \mathsf{SN}$ and all its key reductions are in D, so the term is in D. Therefore, $\{\overline{s} \,;\, \overline{\lambda x.t}\}_r \in \Phi$.

- Suppose $\Phi = c(A_1, \ldots, A_n)$ and

$$\frac{\Gamma \vdash t : \Phi \qquad \ldots \Gamma \vdash p_k : A_k \ldots \qquad \ldots \Gamma, y_\ell : A_\ell \vdash q_\ell : D}{\Gamma \vdash t \cdot_r [\overline{p} \,;\, \overline{\lambda y.q}] : D}\,\text{el}$$

Then $t \cdot_r [\overline{p} \,;\, \overline{\lambda y.q}] = t \cdot_r [\overline{p} \,;\, \overline{\lambda y.q}] \in D$ by $t \in \Phi = c(A_1, \ldots, A_n)$ and the definition of $c(A_1, \ldots, A_n)$ as a saturated set and the induction hypothesis. ◀

The following is now an immediate corollary by taking $\rho(x) := x$ for all $x \in \mathsf{Var}$. Because $\mathsf{Var} \subseteq \mathsf{Neut} \subseteq \langle A \rangle$, we know that $\rho \models \Gamma$. So, if $\Gamma \vdash t : A$, then $\langle t \rangle_\rho = t \in \langle A \rangle \subseteq \mathsf{SN}$.

▶ **Theorem 62.** *The reduction $\longrightarrow_a$ is strongly normalizing: all $\longrightarrow_a$-reductions on proof terms are finite.*

▶ **Corollary 63.** *The reduction $\longrightarrow_a$ for the optimized rules of Definition 36, the parallel $\wedge$-elimination rule of Definition 44, the standard $\to$-introduction of Definition 47 and the traditional rule for $\neg$-elimination of Definition 50 are strongly normalizing.*

Proof. By Theorem 62 and the fact that reduction is preserved by the translation: Lemmas 43, 46 and 49. ◀

6.2 Weak Normalization of conversion

We now give a strategy for finding a normal form for the combined $\longrightarrow_{ab}$ reduction, the union of $\longrightarrow_a$ and $\longrightarrow_b$. This proves that $\longrightarrow_{ab}$ is weakly normalizing and it also gives a concrete procedure for finding a normal form. Due to the fact that, in general, reduction is not confluent, this normal form is not unique, but it does yield *decidability* via the *sub-formula property*. The weak normalization proof follows the well-known idea, originally due to Turing (see [5]) for simple type theory, to *contract the innermost redex of highest rank.*

▶ **Definition 64.** We define the *rank of a formula A*, $\mathsf{rk}(A)$ as follows.

- $\mathsf{rk}(A) := 1$ if A is a proposition letter.
- $\mathsf{rk}(c(A_1, \ldots, A_n) := 1 + \max\{\mathsf{rk}(A_1), \ldots, \mathsf{rk}(A_n)\}$ if c is a connective of arity n.

We define the *rank of a redex* as follows.

- The rank of $\{\overline{p} \,;\, \overline{\lambda x.q}\}_{r'} \cdot_r [\overline{s} \,;\, \overline{\lambda y.r}]$ is the rank of the type of $\{\overline{p} \,;\, \overline{\lambda x.q}\}_{r'}$.
- The rank of $(t \cdot_{r'} [\overline{p} \,;\, \overline{\lambda x.q}]) \cdot_r [\overline{s} \,;\, \overline{\lambda y.r}]$ is the rank of the type of $t \cdot_{r'} [\overline{p} \,;\, \overline{\lambda x.q}]$.

We will sometimes mark the redex with its type Φ such that $\mathsf{rk}(\Phi)$ is the rank of the redex. We do this by writing Φ as a superscript to the elimination constructor. To clarify, we summarize again the possible reduction steps of the form $\longrightarrow_a$ and $\longrightarrow_b$.

▶ **Notation 65.** *From Definition 29, we have the reduction $\longrightarrow_a$ and form Definition 32 we have the reduction $\longrightarrow_b$. We introduce the following notation.*

$$\begin{array}{rcl}
\{\overline{p, p_j} \,;\, \overline{\lambda x.q}\} \cdot^\Phi [\overline{s} \,;\, \overline{\lambda y.r, \lambda y_\ell.r_\ell}] & \longrightarrow_{a1} & r_\ell[y_\ell := p_j] \\
\{\overline{p} \,;\, \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^\Phi [\overline{s, s_k} \,;\, \overline{\lambda y.r}] & \longrightarrow_{a2} & q_i[x_i := s_k] \cdot^\Phi [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \\
(t \cdot [\overline{p} \,;\, \overline{\lambda x.q}]) \cdot^\Phi [\overline{s} \,;\, \overline{\lambda y.r}] & \longrightarrow_b & t \cdot [\overline{p} \,;\, \overline{\lambda x.(q \cdot^\Phi [\overline{s} \,;\, \overline{\lambda y.r}])}]
\end{array}$$

Here, the proviso's of Definition 29 apply, so the first is the "$\ell = j$ case" which we will call $\longrightarrow_{a1}$, and the second is the "$k = i$ case" which we will call $\longrightarrow_{a2}$.

We give two Lemmas that show that the creation of new redexes is limited.

▶ **Lemma 66.** **1.** *If $t \longrightarrow_b t'$ by contracting a redex of $\mathsf{rk}(\Phi)$ then the newly created redexes are also of $\mathsf{rk}(\Phi)$.*

2. *Suppose $\{\overline{p}\ ;\ \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}]$. If $q_i[x_i := s_k]$ is an introduction term (that is: $q_i[x_i := s_k]$ is of the form $\{\ldots\ ;\ \ldots\}$), then q_i is an introduction term. Similarly, if $q_i[x_i := s_k]$ is an elimination term (that is: $q_i[x_i := s_k]$ is of the form $\ldots \cdot [\ldots\ ;\ \ldots]$), then q_i is an elimination term.*

Proof. **1.** If $t \longrightarrow_b t'$ by contracting a redex of $\mathsf{rk}(\Phi)$, then t contains a sub-term $s \cdot [\overline{p}\ ;\ \overline{\lambda x.q}] \cdot^{\Phi} [\overline{u}\ ;\ \overline{\lambda y.r}]$ which is contracted to $s \cdot [\overline{p}\ ;\ \overline{\lambda x.q \cdot^{\Phi} [\overline{u}\ ;\ \overline{\lambda y.r}]}]$. The newly created redexes (if any) are all of $\mathsf{rk}(\Phi)$.

2. Suppose $\{\overline{p}\ ;\ \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}]$. Then $q_i : \Phi$ and $s_k : A_k$ which is a sub-formula of Φ, as $\Phi = c(A_1, \ldots, A_n)$. If $q_i[x_i := s_k]$ is an introduction term, then either q_i is an introduction term itself or $q_i = x_i$ and s_k is an introduction term. The latter case can only occur if $s_k : \Phi$, but it is not, because its type is a sub-formula of Φ. So q_i is an introduction term. The case for $q_i[x_i := s_k]$ being an elimination term is similar. ◀

The Lemma states that both the newly created redexes due to $\longrightarrow_b$ and $\longrightarrow_{a2}$ are already "hidden" inside the term. We give a list of facts about redex creation and the ranks of redexes.

▶ **Fact 67.** **1.** *A reduction step can produce more redexes either by (i)* copying existing redexes *or by (ii)* creating new redexes. *Copying occurs through substitution, in a reduction step $\longrightarrow_{a1}$ or $\longrightarrow_{a2}$.*

2. *Creating new redexes happens in either one of the following ways.*

a. *When doing an $\longrightarrow_a$ step: in a sub-term $x \cdot [\overline{p}\ ;\ \overline{\lambda y.q}]$, we substitute $\{\overline{s}\ ;\ \overline{\lambda z.r}\}$ for x. This creates an a-redex of lower rank.*

b. *When doing an $\longrightarrow_a$ step: in a sub-term $x \cdot [\overline{p}\ ;\ \overline{\lambda y.q}]$, we substitute $t \cdot [\overline{s}\ ;\ \overline{\lambda z.r}]$ for x This creates a b-redex of lower rank.*

c. *When $\{\overline{p, p_j}\ ;\ \overline{\lambda x.q}\} \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r, \lambda y_\ell.r_\ell}] \longrightarrow_{a1} r_\ell[y_\ell := p_j]$ where this term occurs as a sub-term: $r_\ell[y_\ell := p_j] \cdot^{\Psi} [\ldots\ ;\ \ldots]$ and $r_\ell[y_\ell := p_j] = \{\ldots\ ;\ \ldots\}$. This creates a new a-redex of unrelated rank.*

d. *When $\{\overline{p, p_j}\ ;\ \overline{\lambda x.q}\} \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r, \lambda y_\ell.r_\ell}] \longrightarrow_{a1} r_\ell[y_\ell := p_j]$ where this term occurs as a sub-term: $r_\ell[y_\ell := p_j] \cdot^{\Psi} [\ldots\ ;\ \ldots]$ and $r_\ell[y_\ell := p_j] = \ldots \cdot [\ldots\ ;\ \ldots]$. This creates a new b-redex of unrelated rank.*

e. *When $\{\overline{p}\ ;\ \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}]$, where $q_i = \{\ldots\ ;\ \ldots\}$. This creates a new a-redex of the same rank.*

f. *When $\{\overline{p}\ ;\ \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k}\ ;\ \overline{\lambda y.r}]$, where $q_i = \ldots \cdot [\ldots\ ;\ \ldots]$. This creates a new b-redex of the same rank.*

g. *If $(t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}]) \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r}] \longrightarrow_b t \cdot [\overline{p}\ ;\ \overline{\lambda x.(q \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r}])}]$, where $q_i = \{\ldots\ ;\ \ldots\}$. This creates a new a-redex (possibly more) of the same rank.*

h. *If $(t \cdot [\overline{p}\ ;\ \overline{\lambda x.q}]) \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r}] \longrightarrow_b t \cdot [\overline{p}\ ;\ \overline{\lambda x.(q \cdot^{\Phi} [\overline{s}\ ;\ \overline{\lambda y.r}])}]$, where $q_i = \ldots \cdot [\ldots\ ;\ \ldots]$. This creates a new b-redex (possibly more) of the same rank.*

Note that in the cases **e** and **f** of Fact 67 we use the second part of Lemma 66.

The idea is to contract an innermost redex of highest rank of a term in b-normal form (that is: a term that cannot do a $\longrightarrow_b$-step). The advantage of b-normal forms is that cases **c** and **d** of the Fact 67 do not occur. (Because in these cases, the term one starts with is not in b-normal form.)

▶ **Lemma 68.** *If f is a well-typed term in b-normal form that has one redex of maximum rank, say R, then f can be reduced to a term f' in b-normal form that has maximum rank below R.*

Proof. By induction on the size of f.

1. If $f = \{\overline{p} \,;\, \overline{\lambda x.q}\}$ or $f = x \cdot [\overline{p} \,;\, \overline{\lambda x.q}]$ or $f = \{\overline{p} \,;\, \overline{\lambda x.q}\} \cdot [\overline{s} \,;\, \overline{\lambda y.r}]$ and the redex of highest rank is inside $\overline{p}$, $\overline{q}$, $\overline{s}$ or $\overline{r}$, then we are done by the induction hypothesis.
2. Suppose $f = \{\overline{p} \,;\, \overline{\lambda x.q}\} \cdot^{\Phi} [\overline{s} \,;\, \overline{\lambda y.r}]$ is itself a redex of highest rank, $\mathsf{rk}(\Phi)$. We look at the possible ways in which a new redex may arise, following Fact 67. The cases **c**, **d**, **g** and **h** don't apply.
   - For case **a**: the newly created redexes are of lower rank and the resulting term is in b-nf.
   - For case **b**: the newly created redexes are of lower rank. The resulting term may not be in b-nf, but we can contract all the newly created b-redexes to obtain a b-normal form. According to Lemma 66, case (1), this does not create new redexes of higher rank, so we are done.
   - For case **e**: $f = \{\overline{p} \,;\, \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$ with $q_i = \{\ldots \,;\, \ldots\}$. By induction hypothesis, $q_i \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \twoheadrightarrow g$ for some g in b-normal form with all redexes of lower rank. (Note that $q_i \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$ is in b-normal form.) Then $q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \twoheadrightarrow g[x_i := s_k]$ and due to the fact that the type of s_k is a sub-formula of Φ, this only contains new redexes of lower rank, so we are done.
   - For case **f**: $f = \{\overline{p} \,;\, \overline{\lambda x.q, \lambda x_i.q_i}\} \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$ with $q_i = t \cdot [\overline{u} \,;\, \overline{\lambda z.v}]$. If we take g to be the b-normal form of $q_i \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$, this term contains disjoint sub-terms of the shape $\lambda w.d \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$ that all have one maximal redex of rank R and that have length smaller than the length of f. By induction hypothesis, these can all be reduced to terms with only redexes of lower rank. Having done this, we obtain g as a reduct of $q_i \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}]$ that is in b-normal form and contains only redexes of rank lower than R. To obtain f', we notice that $f \longrightarrow_{a2} q_i[x_i := s_k] \cdot^{\Phi} [\overline{s, s_k} \,;\, \overline{\lambda y.r}] \twoheadrightarrow g'[x_i := s_k]$, which only contains b-redexs of lower rank, so we can take f' to be the b-normal form of $g'[x_i := s_k]$. ◀

▶ **Theorem 69.** *For any set of connectives $\mathcal{C}$, the reduction $\longrightarrow_{ab}$ of the calculus $\lambda^{\mathcal{C}}$ is weakly normalizing and we have a procedure to compute a normal form for a well-typed term.*

Proof. We consider the following measure $\mathsf{m}(-)$ terms: $\mathsf{m}(t) := (R, m)$, where R is the maximal rank of a redex in t and m is the number of redexes of rank R in t. We consider this measure under the lexicographic ordering.

Given a term t, we first compute its b-normal form, t_1 and consider $\mathsf{m}(t_1) = (R, m)$. Then we pick p, an innermost redex of maximal rank inside t_1. Following Lemma 68, we reduce p to p', in which all redexes are of rank below R. We do this reduction on t_1, obtaining t_2. (So $t_1 \twoheadrightarrow t_2$.) Notice that $\mathsf{m}(t_1) > \mathsf{m}(t_2)$. We continue in this way, obtaining a normal form of t, because the lexicographic ordering is well-founded. ◀

We recall Lemma 35 which describes NF inductively, the set of terms in normal form. If t is in normal form, then t is of either one of the following three forms

1. t is a variable,
2. $t = \{\overline{p} \,;\, \overline{\lambda y.q}\}$, with all p_i and q_j in normal form,
3. $t = x \cdot [\overline{p} \,;\, \overline{\lambda y.q}]$, with x a variable and all p_i and q_j in normal form.

6.3 Corollaries of normalization

▶ **Theorem 70.** *For any set of connectives $\mathcal{C}$, the calculus $\lambda^{\mathcal{C}}$ is consistent, that is: there are types A for which there is no closed term t with $\vdash t : A$.*

Proof. Take A to be a propositional variable and suppose $\vdash t : A$ with t in normal form. The three possible cases for t are given in Lemma 35, which we have recalled above. The first and third case are impossible, because t cannot contain any free variable. The second case is impossible, because an introduction term is always of a composite type. ◀

The calculus (and logic) $\lambda^{\mathcal{C}}$ also satisfies the sub-formula property.

▶ **Theorem 71.** *Given a set of connectives $\mathcal{C}$, the calculus $\lambda^{\mathcal{C}}$ satisfies the* sub-formula property*, that is: if $\Gamma \vdash t : A$, then there is a term t' such that $\Gamma \vdash t' : A$ and all types of all sub-terms of t' are either sub-types of A or of some A_i for a declaration $x_i : A_i$ in Γ.*

Proof. If $\Gamma \vdash t : A$, then (by Theorem 69) there is a term t' in normal form with $\Gamma \vdash t' : A$. We use Lemma 35 and prove by induction on t' that "all types of all sub-terms of t' are either sub-types of A or of some A_i for a declaration $x_i : A_i$ in Γ". For simplicity we abbreviate this property to "*t' satisfies the sub-type property for* $\Gamma; A$".

- $t' = x$, a variable. Then we are done.
- $t' = \{\overline{p} \;;\; \overline{\lambda x.q}\}$, an introduction term. Then by induction hypothesis, all sub-terms of $\overline{p}$ satisfy the sub-type property for $\Gamma; A_i$ for some A_i which is a sub-type of A. For the $\lambda x_j.q_j$ in $\overline{\lambda x.q}$, we have $\Gamma, x_j : A_j \vdash q_j : A$ for some A_j which is a sub-type of A. By induction hypothesis, for all j, all sub-terms of q_j satisfy the sub-type property for $\Gamma, x_j : A_j; A$. So all sub-terms of $\overline{\lambda x.q}$ satisfy the sub-type property for $\Gamma; A$ and we are done.
- $t' = x \cdot [\overline{p} \;;\; \overline{\lambda x.q}]$, an elimination term. Suppose $x : C$. Each p_i is of type B_i for some sub-type B_i of C, so the induction hypothesis yields that all sub-terms of $\overline{p}$ satisfy the sub-type property for $\Gamma; A$. For the $\lambda x_j.q_j$ in $\overline{\lambda x.q}$, we have $\Gamma, x_j : B_j \vdash q_j : A$ for some B_j which is a sub-type of C. By induction hypothesis, for all j, all sub-terms of q_j satisfy the sub-type property for $\Gamma, x_j : B_j; A$. So all sub-terms of $\overline{\lambda x.q}$ satisfy the sub-type property for $\Gamma; A$ and we are done. ◀

▶ **Theorem 72.** *In $\lambda^{\mathcal{C}}$, given a context Γ and a type D, the problem $\Gamma \vdash ? : D$ is decidable. That is, it is whether there is a term t for which $\Gamma \vdash t : D$.*

Proof. By Theorem 69 we can limit our search to a term in normal form. So we can restrict the elimination rules to the following restricted case, where $\Phi = c(A_1, \ldots, A_n)$. (Compare with the original rules in Definition 27.)

$$\frac{x : \Phi \in \Gamma \quad \ldots \Gamma \vdash p_k : A_k \ldots \quad \ldots \Gamma, y_\ell : A_\ell \vdash q_\ell : D}{\Gamma \vdash x \cdot_r [\overline{p} \;;\; \overline{\lambda y.q}] : D}\,\text{el}$$

Now, given Γ and D, the following algorithm searches a term t in normal form with $\Gamma \vdash t : D$. (1) Check if $x : D \in \Gamma$ for some x and otherwise (2) try an introduction rule (in case D is composite) and (3) try an elimination rule for each $x : \Phi \in \Gamma$ with Φ a composite formula. In the recursive case, this gives finitely many possibilities to try and each try creates new goals of the form $\Gamma, y_j : A_j \vdash ? : D$ or of the form $\Gamma \vdash ? : A_i$ with A_j and A_i sub-formulas of Γ, D. This search terminates because the number of sub-formulas in the context increases (which is bound by the number of all sub-formulas of Γ, D), and otherwise the size of the goal-formula decreases. ◀

As a corollary, we find that all the variants of the logical rules we have considered are decidable and consistent, simply because they are (with respect to derivability) equivalent to the set of rules for $\wedge, \vee, \to, \neg, \bot, \top$ that we extract from the truth tables, for which Theorems 70 and 72 apply. We can also say a bit more about the conversion of derivations in these systems themselves: detour conversion is strongly normalizing, permutation conversion is strongly normalizing and we can also conclude weak normalization of the combined conversion.

▶ **Theorem 73.** *The reductions for the optimized rules of Definition 36, the parallel $\wedge$-elimination rule of Definition 44, the standard $\to$-introduction of Definition 47 and the traditional rule for $\neg$-elimination of Definition 50 are weakly normalizing.*

Proof. The proof follows the same argument as the proof of Theorem 69. The crucial Lemmas are Lemmas 68 and 66, which can be proved again with the reduction rules mentioned in the statement of Theorem 73 added. Furthermore, the permutation conversion, $\longrightarrow_b$ is strongly normalizing. (Corollary 56.) ◀

7 Conclusion and Further work

We have studied the general procedure for deriving intuitionistic natural deduction rules from truth tables, that we have presented in [7]. We have defined detour conversion and permutation in general and we have proven that both are strongly normalizing and that the combination of the two is weakly normalizing. We have done so by defining a proof-term calculus for derivations on which we have defined the reduction rules that correspond to conversion of derivations. This follows the well-known Curry-Howard formulas-as-types isomorphism that establishes an isomorphism between proofs (derivations in natural deduction) and terms. We have shown that very many well-known formalisms for intuitionistic natural deduction can be defined in terms of our calculus, including the conversion rules for derivations. Our paper also provides a straightforward method for deriving a term calculus for any connective that is given via a truth table: the term constructions and reduction rules are self-contained and normalizing by construction. We have shown this on various examples, most notably the nand-connective.

The work described here leaves various questions unanswered. For example, is proof normalization (the combination of detour conversion and permutation conversion) strongly normalizing in general for an arbitrary set of connectives? We would believe so, but have not yet proved it. Techniques as in [9], where this property is proved for intuitionistic logic, may be useful.

It also raises various new research questions: The rules are not Church-Rosser (confluent) in general, but one may wonder whether there is a certain condition that guarantees confluence. We have seen in Examples 23, 30 and 40 that fixing a choice for the "matching case" in a detour convertibility may render the reduction confluent. It is not clear if this would work in general.

Another topic to look into is detour conversion for the classical case, and what its connection is with known term calculi for classical logic, for example as studied in [13], [1] and [2]. Also, it might be interesting to look at these general rules from a linear perspective: what if we enforce the rules to be linear?

Finally, we may wonder whether our research could contribute to the study of "harmony in logic", as first introduced by Prawitz [15] and further studied by various authors like [16, 12, 23, 4, 3]. The inversion principle explains the elimination rules as capturing the "least information" that is conveyed by the introduction rules. This can also be dualized (as is done in [12] in their “uniform calculus”) by explaining the introduction rules in terms of the elimination rules. It would be interesting to study the relation with our rules, where there is no a priori preference for the introduction or elimination rules.

From our research, we would propose the following as a proper system for intuitionistic logic with “parallel elimination rules” that follow Prawitz’ [15] inversion principle. These rules are derived from the truth tables and optimized following Lemma 9, but not using Lemma 12. Compare with Definition 13; the special rules are $\wedge$-elimination and $\to$-elimination.

▶ **Definition 74.** The *parallel elimination rules* for the intuitionistic propositional connectives $\wedge, \vee, \to, \neg, \bot$ and $\top$ are given below.

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

$$\frac{\vdash A}{\vdash A \vee B}\,\vee\text{-inl} \qquad \frac{\vdash B}{\vdash A \vee B}\,\vee\text{-inr} \qquad \frac{\vdash A \vee B \quad A \vdash D \quad B \vdash D}{\vdash D}\,\vee\text{-el}$$

$$\frac{A \vdash A \to B}{\vdash A \to B}\,\to\text{-in}_a \qquad \frac{\vdash B}{\vdash A \to B}\,\to\text{-in}_b \qquad \frac{\vdash A \to B \quad \vdash A \quad B \vdash D}{\vdash D}\,\to\text{-el}$$

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

References

1 Z. Ariola and H. Herbelin. Minimal Classical Logic and Control Operators. In *ICALP*, volume 2719 of *LNCS*, pages 871–885. Springer, 2003.

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

3 R. Dyckhoff. Some Remarks on Proof-Theoretic Semantics. In *Advances in Proof-Theoretic Semantics*, pages 79–93. Springer, 2016.

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

5 R.O. Gandy. An early proof of normalization by A.M. Turing. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, page 453–455. Academic Press Limited, 1980.

6 G. Gentzen. Untersuchungen über das logische Schliessen. *Mathematische Zeitschrift*, pages 176–210, 405–431, 1935. English translation in [19].

7 H. Geuvers and T. Hurkens. Deriving Natural Deduction Rules from Truth Tables. In *ICLA*, volume 10119 of *Lecture Notes in Computer Science*, pages 123–138. Springer, 2017.

8 J.-Y. Girard et al. *Proofs and types*, volume 7 of *Cambridge tracts in theoretical computer science.* Cambridge University Press, Cambridge, 1989.

9 F. Joachimski and R. Matthes. Short proofs of normalization for the simply- typed lambda-calculus, permutative conversions and Gödel’s T. *Arch. Math. Log.*, 42(1):59–87, 2003.

10 E.G.K López-Escobar. Standardizing the N systems of Gentzen. In *Models, Algebras and Proofs*, volume 203 of *Lecture Notes in Pure and Applied Mathematics*, page 411–434. Marcel Dekker Inc., New York, 1999.

11 P. Milne. Inversion Principles and Introduction Rules. In *Dag Prawitz on Proofs and Meaning*, volume 7 of *Outstanding Contributions to Logic*, pages 189–224. Springer, 2015.

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

13 M. Parigot. $\lambda\mu$-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *LPAR*, volume 624 of *LNCS*, pages 190–201. Springer, 1992.

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

15 D. Prawitz. Ideas and Results in Proof Theory. In J. Fenstad, editor, *2nd Scandinavian Logic Symposium*, pages 237–309. North-Holland, 1971.

16 S. Read. Harmony and Autonomy in Classical Logic. *J. Philosophical Logic*, 29(2):123–154, 2000. `doi:10.1023/A:1004787622057`.

17 P. Schroeder-Heister. A Natural Extension of Natural Deduction. *J. Symb. Log.*, 49(4):1284–1300, 1984.

18 Morten Heine Sørensen and Pawel Urzyczyn. *Lectures on the Curry-Howard Isomorphism, Volume 149 (Studies in Logic and the Foundations of Mathematics).* Elsevier Science Inc., New York, NY, USA, 2006.

19 M.E. Szabo. *The Collected Papers of Gerhard Gentzen.* North-Holland, Amsterdam, 1969.

20 W.W. Tait. Intensional Interpretations of Functionals of Finite Type I. *J. Symb. Log.*, 32(2):198–212, 1967. `doi:10.2307/2271658`.

21 N. Tennant. Ultimate Normal Forms for Parallelized Natural Deductions. *Logic Journal of the IGPL*, 10(3):299–337, 2002.

22 D. van Dalen. *Logic and structure (3. ed.).* Universitext. Springer, 1994.

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

PML_2: Integrated Program Verification in ML

Rodolphe Lepigre
LAMA, CNRS, Université Savoie Mont Blanc, France &
Inria, LSV, CNRS, Université Paris-Saclay, France
rodolphe.lepigre@inria.fr
https://orcid.org/0000-0002-2849-5338

Abstract

We present the PML_2 language, which provides a uniform environment for programming, and for proving properties of programs in an ML-like setting. The language is Curry-style and call-by-value, it provides a control operator (interpreted in terms of classical logic), it supports general recursion and a very general form of (implicit, non-coercive) subtyping. In the system, equational properties of programs are expressed using two new type formers, and they are proved by constructing terminating programs. Although proofs rely heavily on equational reasoning, equalities are exclusively managed by the type-checker. This means that the user only has to choose which equality to use, and not where to use it, as is usually done in mathematical proofs. In the system, writing proofs mostly amounts to applying lemmas (possibly recursive function calls), and to perform case analyses (pattern matchings).

2012 ACM Subject Classification Software and its engineering → Software verification

Keywords and phrases program verification, classical logic, ML-like language, termination checking, Curry-style quantification, implicit subtyping

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.4

1 Introduction: joining programming and proving

In the last thirty years, significant progress has been made in the application of type theory to computer languages. The Curry-Howard correspondence, which links the type systems of functional programming languages to mathematical logic, has been explored in two main directions. On the one hand, proof assistants such as Agda [24] or Coq [23] are based on very expressive logics [22, 7]. To establish their consistency, the underlying programming languages need to be restricted to provably terminating programs. As a result, they forbid the most general forms of recursion. On the other hand, functional programming languages such as Haskell, SML or OCaml are well-suited for programming, as they impose no restriction on recursion. However, their type systems are inconsistent when considered as logics, which means that they cannot be used for proving mathematical formulas.[1]

The aim of PML_2 is to provide a uniform environment in which programs can be designed, specified and proved. The idea is to combine a full-fledged ML-like programming language, with an enriched type system allowing the specification of computational behaviours.[2] The obtained system can thus be used as ML for type-safe general programming, and as a proof assistant for proving properties of ML programs. The uniformity of the framework implies that programs can be incrementally refined to obtain more and more guarantees. In particular, there is no syntactic distinction between programs and proofs. The only difference is that the

[1] This particular point will be explained in more detail in Section 5.
[2] On might argue that PML_2 is not a *full-fledged ML-like language* as it does not have mutable references. It is nonetheless effectful as it provides a control operator similar to Scheme's *call/cc*.

latter must be typed-checked against the consistent core of the system, which only accepts programs that can be proved terminating. In the current implementation, programs must also be proved terminating to be directly accepted by the type-checker. It would however be possible to accept programs that do not pass the termination check, and we would then need to make sure that such programs are not used to write proofs.[3] Note however that the system can already be used to reason about arbitrary programs, including untyped ones and those whose termination cannot be established (an example will be given in Section 5.3).

1.1 Program verification principles

In PML$_2$, program properties may be specified with types containing equations of the form `t ≡ u`, where `t` and `u` are terms of the language itself. By quantifying over the free variables of these terms, we can express properties such as the following.

```
// "add" is commutative.
∀n∈nat, ∀m∈nat, add n m ≡ add m n

// "reverse" is involutive.
∀a, ∀l∈list⟨a⟩, reverse (reverse l) ≡ l

// "sort" produces sorted lists.
∀l∈list⟨nat⟩, sorted (sort l) ≡ true

// All natural numbers are equal to "Zero".
∀n∈nat, n ≡ Zero.

// Sorted lists are not affected by "sort".
∀l∈list⟨nat⟩, (sorted l ≡ true) ⇒ (sort l ≡ l)
```

Of course, such a specification may be inaccurate, in which case it will not be provable. Note that it is possible to observe complex behaviours using predicates such as `sorted`, which correspond to boolean-valued functions.

The PML$_2$ language relies on two main ingredients for proving that programs meet their specifications. First, the type system of the language can be considered as a (classical) logic through the Curry-Howard correspondence, although it is only consistent for terminating programs. This (usual) part of the type system provides basic reasoning principles, which are used to structure the proofs. The second ingredient is an automatic decision procedure for the equational theory of the language. It is used to manage a context of equational assumptions, and eventually prove equations in this context. The decision procedure is driven by the type-checker, without any direct interaction with the user. As a consequence, the user only has to care about the structure of the proof, and not about the details of where equations should be applied. In fact, equality types of the form `t ≡ u` are computationally irrelevant in the system. More precisely, `t ≡ u` is equivalent to the unit type when the denoted equality holds (and can be proved), and it is empty otherwise. As a consequence, a proof will generally consist of a (possibly recursive) program that calls other programs and performs pattern matching but eventually just returns a completely uninteresting result. Nonetheless, writing proofs in this way is a very similar experience to writing functional programs. We can hence hope that our approach to program verification will feel particularly intuitive to functional programmers.

[3] For example, we could have two different function types: one used for functions whose termination has been established, and another one that that may be used for any function (and that would be a supertype of the former).

1.2 Previous work on the language

PML$_2$ is based on many ideas introduced by Christophe Raffalli in the PML language [27]. Although this first version of the system was very encouraging on the practical side, it did not stand on solid theoretical grounds. On the contrary, PML$_2$ is based on a call-by-value, classical realizability model designed by the author [16]. This framework provides a satisfactory way of combining call-by-value evaluation and effects with dependent function types.[4] The proposed solution relies on a relaxed form of value restriction (called *semantical value restriction*), which takes advantage of our notion of program equivalence and its decision procedure.[5] In particular, it allows the application of a dependent function to a term which is not a value, under the condition that it can be proved equivalent to some value. This is especially important because dependent functions are an essential component of PML$_2$. Indeed, they enable a form of typed quantification without which many program properties could not be expressed (see Section 3).

Another important specificity of PML$_2$'s type system is that it relies on the notion of *local subtyping*, which was introduced in joint work with Christophe Raffalli [19]. This framework can be used to give a syntax-directed formulation of the typing and subtyping rules of the system[6], despite its Curry-style nature. In particular, it provides a very general notion of infinite (circular) proof, that is used for handling inductive and coinductive types in subtyping derivations, and recursion and termination checking in typing derivations. Of course, infinite proofs are only valid if they are well-founded (this is ensured using the size-change principle [15]). The combination of local subtyping [19] and the realizability model of the system has been addressed in the author's PhD thesis [17, Chapter 6].

Last but not least, the implementation of PML$_2$ [18] was initiated as part of the author's thesis [17], and continues with the collaboration of Christophe Raffalli. The implemented system is intended to remain very close to the theoretical type system, and every part of the implementation is justified by the formal semantics of PML$_2$ [17]. Note that all the examples given in this paper are accepted by the version `2.0.2_types2017` of PML$_2$, which can be downloaded at the following URL.

`https://github.com/rlepigre/pml/archive/pml_2.0.2_types2017.tar.gz`

1.3 Disclaimer: the aim of this paper

This document is intended to be an introductory paper for the PML$_2$ system. Its aim is not to give the details of the realizability semantics, nor to prove new theoretical results, but rather to list the principles and ideas on which PML$_2$ is based. In particular, Section 6 contains an extensive description of several ideas that we would like to investigate in the near future, and that will be necessary for achieving the goals of PML$_2$ completely. For technical details, the reader should refer to the author's thesis [17], and related papers [16, 19].

[4] Due to the soundness issues explained in previous work [16], the application of dependent functions is usually restricted to value arguments. This is commonly called *value restriction* in the context of ML.

[5] Intuitively, two terms are (observationally) equivalent if they have the same computational behaviour (i.e., they both converge or they both diverge) in every possible evaluation context [16, 17].

[6] This means that exactly one typing rule applies for every term constructor, and only one subtyping rule applies for every pair of type constructors (up to commutation and circular proof construction).

2 Functional programming in PML$_2$

Our first goal in designing PML$_2$ was to obtain a practical, functional programming language. Out of the many possible technical choices, we decided to consider a call-by-value language similar to OCaml or SML, as they have proved to be highly practical and efficient. Our language provides polymorphic variants [8] and SML-style records, which are convenient for encoding data types. As an example, the type of lists can be defined as follows,[7] together with the corresponding `iter` and `append` functions.

```
type rec list⟨a⟩ = [Nil ; Cns of {hd : a ; tl : list}]

val rec iter : ∀a, (a ⇒ {}) ⇒ list⟨a⟩ ⇒ {} =
  fun f l {
    case l {
      Nil    → {}
      Cns[c] → f c.hd; iter f c.tl
    }
  }

val rec append : ∀a, list⟨a⟩ ⇒ list⟨a⟩ ⇒ list⟨a⟩ =
  fun l1 l2 {
    case l1 {
      Nil    → l2
      Cns[c] → Cns[{hd = c.hd ; tl = append c.tl l2}]
    }
  }
```

Note that the `iter` and `append` functions are polymorphic, which means, for instance, that they can be applied to lists with elements of an arbitrary type. In our syntax, this is explicitly materialised using universally quantified type variables. Note also that the `iter` function relies on the type `{}`, which contains records with no fields. It plays the same role as OCaml's `unit` type, and its unique inhabitant is denoted `{}` as well.

▶ Remark. As in System F [9, 28], polymorphism can be used anywhere in types, and it is not limited to *let-polymorphism* (or prenex polymorphism) as in most ML-like languages.

2.1 Control operator and classical logic

The programming languages of the ML family generally include effectful operations such as references (i.e., mutable variables). Our system is no exception since it provides a control operator similar to *call/cc*.[8] On the programming side, it may be used to encode a form of exception mechanism. For instance, we can define the following `exists` function, which tests whether there is an element satisfying a given predicate in a given list, and stops as soon as possible if such an element is found.

[7] Note that `list` is used without its type parameter in the type of the `Cns` constructor. This is due to the fact that the "`type rec`" syntax is desugared to an inductive type (or least fixed point), and `list`⟨`a`⟩ is actually defined as "μ `list, [Nil; Cns of {hd : a; tl : list}]`". In particular, the current version of PML$_2$ does not support polymorphically recursive types.

[8] This instruction can be used to capture the current continuation (or evaluation context), so that it can be restored later. It was first introduced in the Scheme language.

```
val exists : ∀a, (a ⇒ bool) ⇒ list⟨a⟩ ⇒ bool =
  fun pred l {
    save k {
      iter (fun e { if pred e { restore k true } }) l;
      false
    }
  }
```

Here, the continuation is saved in a variable `k` before calling the `iter` function, and it is restored with the value `true` if an element satisfying the predicate is found. In this case, the evaluation of `iter` is simply aborted. To obtain a similar behaviour without using a continuation would require the user to write an independent recursive function (i.e., one that does not rely on `iter`). A more interesting example that cannot be written without a control operator will be given in Section 4.

As is now well-known, control operators such as ours can be used to give a computational content to classical theorems, thus extending the Curry-Howard correspondence to classical logic [10]. It is hence possible to define programs with a type corresponding to Peirce's law, or to the law of the excluded middle.

```
val peirce : ∀a b, ((a ⇒ b) ⇒ a) ⇒ a =
  fun x {
    save k {
      x (fun y { restore k y })
    }
  }

// Disjoint sum (logical disjunction) and (logical) negation
type either⟨a,b⟩ = [InL of a ; InR of b]
type neg⟨a⟩ = a ⇒ ∀x,x

val excl_mid : ∀a, {} ⇒ either⟨a, neg⟨a⟩⟩ =
  fun _ {
    save k {
      InR[fun x { restore k InL[x] }]
    }
  }
```

Note that the definition of `excl_mid` requires a dummy function constructor due to the call-by-value evaluation strategy. Indeed, `excl_mid` would not be a value if it did not start with an abstraction, and it would thus save its continuation right away, unlike `peirce` which must first be given an argument to trigger the computation. This is related to value restriction [35, 34], which is required in presence of control operators [11].[9]

From a computational point of view, manipulating continuations using control operators can be understood as "cheating". For example `excl_mid` (or rather, `excl_mid {}`) saves the continuation and immediately returns a (possibly false) proof of `neg`⟨a⟩. Now, if this proof is ever applied to a proof of `a` (which would result in absurdity), the program backtracks and returns the given proof of `a`. This interpretation has been well-known for a long time, and an account is given in the work of Wadler [33, Section 4], for example.

[9] Value restriction is a sufficient (but not necessary) condition for correctness.

2.2 Non-coercive subtyping

In PML$_2$, subtyping plays a very important role as it allows us to give a mostly syntax-directed presentation of the system [19, 17]. Although it is less widespread than polymorphism in mainstream languages, subtyping can be exploited to improve code modularity. Note that we here consider a *non-coercive* form of subtyping, which means that if `a` is a subtype of `b`, then any value of type `a` is also a value of type `b` (i.e., no coercion is required).

In the system, there are many forms of subtyping that may interact. In particular, subtyping is used to handle all the connectives that do not have algorithmic contents (i.e., no counterpart in the syntax of terms). Such connectives include quantifiers as well as inductive types, but also the equality types of PML$_2$. Subtyping also plays an important role with variants and records. For instance, it implies that a record can always have more fields than required. Moreover, subtyping enables many commutations of connectives.

As a first example, we can show that the type corresponding to the (classical) double negation elimination principle can in fact be seen as an instance of Peirce's law. Indeed, it can be defined as follows in PML$_2$.

```
val dneg_elim : ∀a, neg⟨neg⟨a⟩⟩ ⇒ a =
  peirce
```

It is relatively easy to see that the type of `peirce` is indeed a subtype of that of `dneg_elim`. The corresponding subtyping derivation is sketched below.[10]

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x} \subseteq \texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}} \quad \dfrac{\dfrac{}{\texttt{a0} \subseteq \texttt{a0}}}{\forall \texttt{x},\texttt{x} \subseteq \texttt{a0}}}{(\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \forall \texttt{x},\texttt{x} \subseteq (\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a0}} \quad \dfrac{}{\texttt{a0} \subseteq \texttt{a0}}}{((\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a0}) \Rightarrow \texttt{a0} \subseteq ((\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a0}}}{\forall \texttt{b}, ((\texttt{a0} \Rightarrow \texttt{b}) \Rightarrow \texttt{a0}) \Rightarrow \texttt{a0} \subseteq ((\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a0}}}{\forall \texttt{a}, \forall \texttt{b}, ((\texttt{a} \Rightarrow \texttt{b}) \Rightarrow \texttt{a}) \Rightarrow \texttt{a} \subseteq ((\texttt{a0} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a0}}}{\forall \texttt{a}, \forall \texttt{b}, ((\texttt{a} \Rightarrow \texttt{b}) \Rightarrow \texttt{a}) \Rightarrow \texttt{a} \subseteq \forall \texttt{a}, ((\texttt{a} \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \forall \texttt{x},\texttt{x}) \Rightarrow \texttt{a}}$$

Intuitively, universal quantification on the right of the inclusion can be eliminated by introducing a fresh constant. On the left, variables that are quantified over can be replaced by anything. As usual, the subtyping rule for handling the arrow type reverses the inclusion between the domains due to the contra-variance of the arrow type.

We will now consider the extension of the type of lists with an additional constructor allowing constant time concatenation. In PML$_2$, the corresponding type of "append lists" can be defined in such a way that it admits the type of regular lists as a subtype.

```
type rec alist⟨a⟩ =
  [Nil ; Cns of {hd : a; tl : alist} ; App of alist × alist]

// Constant time "append" function.
val alist_append : ∀a, alist⟨a⟩ ⇒ alist⟨a⟩ ⇒ alist⟨a⟩ =
  fun l1 l2 { App[(l1,l2)] }
```

Although regular lists are a special case of "append lists", the converse is not true. To transform an "append list" into a list, it is necessary to define the following recursive, flattening function.

[10] The proof does not contain all the necessary information to ensure its validity. The reader should refer to the author's thesis [17, Figure 6.5] to fill in the missing details.

```
val rec alist_to_list : ∀a, alist⟨a⟩ ⇒ list⟨a⟩ =
  fun l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = c.hd ; tl = alist_to_list c.tl}]
      App[c] → append (alist_to_list c.1) (alist_to_list c.2)
    }
  }
```

Another example of extension for a pre-existing type can be obtained by defining the type of red-black trees as a subtype of binary trees. More precisely, a red-black tree can be represented as a tree whose nodes have an extra color field. Of course, the presence of additional information in the form of a new record field does not prevent the use of tree functions such as binary search.

2.3 Toward the encoding of a module system

Despite its many different features, PML$_2$ remains a fairly small system, which can be implemented rather concisely. Its design is based on the principle that every feature should be orthogonal. For instance, there is only one notion of product type in PML$_2$: records. This is not the case in OCaml, for instance, which provides tuples, records, objects, modules, which all have common product type characteristics.

In PML$_2$, modules can be easily encoded using a combination of records for storing the values, functions for building functors, and existentials for type abstraction. However, the implementation does not yet provide a specific syntax for modules. For instance, there is still no way of "opening" a module so that its values are accessible in the scope. It is nonetheless possible to work with the target of the encoding directly. For example, we can define a type corresponding to the signature (or interface) of a simple module providing an abstract representation for the stack data structure with the corresponding operations.[11]

```
type stack_sig = ∃stack: ο → ο,
  { empty : ∀a, stack⟨a⟩;
    push  : ∀a, a ⇒ stack⟨a⟩ ⇒ stack⟨a⟩;
    pop   : ∀a, stack⟨a⟩ ⇒ [None ; Some of a × stack⟨a⟩] }
```

An implementation of this interface can then be defined by giving a corresponding record value. For example, we can implement stacks with lists as follows.

```
val stack_impl : stack_sig =
  { empty = (Nil : ∀a, list⟨a⟩);
    push  = fun e s { Cns[{hd = e; tl = s}] };
    pop   = fun s {
              case s {
                Nil    → None
                Cns[c] → Some[(c.hd, c.tl)]
              }
            } }
```

[11] Here, $o \to o$ corresponds to the sort of types with one type parameter. In particular, o is the sort of types (or propositions), and we will later encounter the sort of program values ι.

Note that we need to give at least some type annotation for the system to know what to instantiate the existential with. This could be done in a more systematic way with a syntax requiring the user to give the intended definition for the `stack` type.

▶ Remark. It is possible to define a dot-projection operation in order to access abstract types, so that it is possible to write `stack_impl.stack` to refer to the type of stacks. More details are given in previous work [19], and in the corresponding implementation.

3 Verification of ML programs

PML$_2$ is not only a programming language, but also a proof assistant focusing on program verification. Its proof mechanism relies on equality types of the form `t ≡ u`, where `t` and `u` are arbitrary (possibly untyped) terms of the language itself. Such an equality type is inhabited by the term `{}`[12] if the denoted equivalence is true, and it is empty otherwise. Equivalences are managed using a partial decision procedure that is driven by the construction of programs. An equational context is maintained by the type checker to keep track of the equational assumptions during the construction of proofs. This context is extended when new equations are learnt (e.g., when a lemma is applied), and an equation is proved by deriving a contradiction (e.g., two different variants that are equated) from its negation.

Terms not only appear in (equality) types, but also play the role of objects in the underlying logic. In particular, they can be quantified over in types, and thus form one particular domain of discourse. In fact, our system is based on a higher-order logic with several atomic sorts (including types and terms), which means that many different kinds of objects can be quantified over (universally and existentially) in our types. We can for example quantify over types with one type parameter (of sort $o \to o$), as in the signature used for the stack module given in the previous section.

3.1 (Un)typed quantification and unary natural numbers

To illustrate the proof mechanism, we will consider very simple examples of proofs on unary natural numbers. Their type is given below, together with the corresponding addition function defined using recursion on its first argument.

```
type rec nat = [Zero ; S of nat]

val rec add : nat ⇒ nat ⇒ nat =
  fun n m {
    case n {
      Zero → m
      S[k] → S[add k m]
    }
  }
```

As a first example, we will show that for all `n` we have `add Zero n ≡ n`. This property is expressed using the type `∀n:ι, add Zero n ≡ n`, and it is proved as follows.[13]

```
val add_Zero_n : ∀n:ι, add Zero n ≡ n =
  {} // immediate
```

[12] Recall that it denotes a record with no fields, or the unique inhabitant of a one-element type.

[13] Here, the domain of the quantification is the set of values of the language, whose sort is ι. It is not limited to natural numbers, and also encompasses booleans and functions for example.

The proof is immediate (i.e., only `{}`) as we have `add Zero n ≡ n` by definition of `add`. Note that this equivalence holds for every value `n`, whether it corresponds to an element of the type `nat` or not. For instance, it can be used to show `add Zero true ≡ true` since the term `add Zero true` evaluates to `true`.

▶ Remark. Here, it is crucial that `n` ranges only over values of the language, as otherwise the definition of add could not be unfolded. Indeed, since we are in call-by-value, it is only possible to effectively apply a function when its arguments are all values.

Let us now show that for every `n` we have `add n Zero ≡ n`. Although this property looks similar to `add_Zero_n`, the following proof is invalid.

```
// val add_n_Zero : ∀n:ι, add n Zero ≡ n =
//   {} // invalid
```

Indeed, the equivalence `add n Zero ≡ n` does not hold when `n` is not a unary natural number. In this case, the computation of `add n Zero` produces a runtime error while that of `n` does not. As a consequence, we need to rely on a form of quantification that only ranges over unary natural numbers. This can be achieved with the type `∀n∈nat, add n Zero ≡ n`, which corresponds to a (dependent) function taking as input a natural number `n` and returning a proof of `add n Zero ≡ n`. This property can then be proved using induction (i.e., using a recursive function) and case analysis (i.e., pattern matching) with the following program.

```
val rec add_n_Zero : ∀n∈nat, add n Zero ≡ n =
  fun n {
    case n {
      Zero → {}
      S[k] → add_n_Zero k
    }
  }
```

If `n` is `Zero`, then we need to show `add Zero Zero ≡ Zero`, which is immediate by definition of `add`. In the case where `n` is `S[k]` we need to show `add S[k] Zero ≡ S[k]`. By definition of `add`, this reduces to `S[add k Zero] ≡ S[k]`. We can then use the induction hypothesis `add_n_Zero k` to learn `add k Zero ≡ k` and conclude the proof.

▶ Remark. The dependent product type (or typed quantification) constructor is not primitive in PML$_2$. It is encoded using a membership type of the form `t∈a` which contains all the elements of type `a` that are equivalent to the term `t` (it can be seen as a form of singleton type). The dependent function type `∀x∈a, b` is then encoded as `∀x:ι, x∈a ⇒ b`, which corresponds to the relativised quantification scheme (see previous work [16, 17]).

It is important to note that, in our system, a program that is considered as a proof needs to go through a termination checker. Indeed, a looping program could be used to prove anything otherwise.[14] For example, the following proof is rejected.

```
// val rec add_n_Zero_loop : ∀n∈nat, add n Zero ≡ n =
//   fun n {
//     add_n_Zero_loop n
//   }
```

It is however easy to see that `add_Zero_n` and `add_n_Zero` are terminating, and hence valid. In the following, we will only consider programs that can be automatically proved terminating by the system.

[14] More details will be given in Section 5.

3.2 Building up an equational context

There are two main ways of learning new equations in the system. On the one hand, when a term `t` is matched in a case analysis, a given branch can only be reached when the corresponding pattern `C[x]` matches. In this case we can extend the equational context with `t ≡ C[x]`. On the other hand, it is possible to invoke a lemma by calling the corresponding function. In particular, this must be done to use the induction hypothesis in proofs by induction like in `add_Zero_n` or the following lemma.

```
val rec add_n_S_m : ∀n m∈nat, add n S[m] ≡ S[add n m] =
  fun n m {
    case n {
      Zero → {}
      S[k] → add_n_S_m k m
    }
  }
```

In this case, the equation corresponding to the conclusion of the used lemma is directly added to the context. Of course, more complex results can be obtained by combining more lemmas. For example, the following proves the commutativity of addition using a proof by induction with `add_n_Zero` and `add_n_S_m`.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero → add_n_Zero m
      S[k] → add_comm k m; add_n_S_m m k
    }
  }
```

▶ Remark. Note that terms can be put in sequence with a semicolon. In the above proof, the recursive call `add_comm k m` is performed first, before calling `add_n_S_m m k`. They are also type-checked in that order, and the corresponding equations are added to the context one after the other as a side-effect to type-checking. Here, the order in which equations are added is not significant (the resulting equational context is the same either way), but that is not always the case (lemmas may require some equations to hold to be applied).

3.3 Detailed proofs using type annotations

Although the above proof of commutativity is perfectly valid, it might not be easy enough to read by a human. This problem arises in most proof assistants. For instance, it is almost impossible to understand a Coq [23] proof without replaying it step by step in a compatible editor. In PML$_2$, it is possible to annotate proofs to highlight the corresponding thought process. For example, we can reformulate `add_comm` as follows.

```
val rec add_comm : ∀n m∈nat, add n m ≡ add m n =
  fun n m {
    case n {
      Zero → show add Zero m ≡ add m Zero using add_n_Zero m; qed
      S[k] → show add k m ≡ add m k using add_comm k m;
             deduce add S[k] m ≡ S[add m k];
             show add S[k] m ≡ add m S[k] using add_n_S_m m k; qed
    }
  }
```

Note that no addition to the system is required for such annotations to be supported, it is only syntactic sugar. For instance, `qed` is a synonym of `{}`, and `show u1 ≡ u2 using p` is translated to `p : u1 ≡ u2`, which amounts to a type coercion.

▶ Remark. Many examples of proofs and programs are provided with the implementation of the system. Each of the examples given here has been automatically checked upon the generation of the document, they are hence correct with respect to the implementation.

3.4 Mixing proofs and programs

We will now see that the programming and the proving features of PML$_2$ can be mixed when constructing proofs or programs. In fact, there is no obvious distinction between the world of the usual programs, and the world of proofs (remember that proofs are programs in PML$_2$). For instance, it is possible to combine proofs with programs for them to transport properties (e.g., addition carrying its own commutativity). This can be achieved using restriction types, which are in fact used to encode equality types. In PML$_2$, the type `a | t ≡ u` is equivalent to `a` if `t ≡ u` is true, and to the empty type otherwise. The type `t ≡ u` is thus encoded as `{} | t ≡ u`, where `{}` is the unit type. Intuitively, the restriction type can be seen as a form of conjunction with no algorithmic contents.

When combined with existential quantification and the membership type, restriction can be used to encode a *set type* syntax similar to that of NuPrl [6]. Indeed, we can define `{x ∈ a | t ≡ u}`, which contains all the elements of type `a` such that `t ≡ u` holds, as `∃x:ι, x∈(a | t ≡ u)`. This provides a very useful scheme for defining the set of terms of `a` that satisfy some property. For example, we can encode the type of vectors (i.e., lists of a given length) by taking every list `l` that has size `s`. The type of vectors will hence have two parameters: the type of the contained elements and a term giving the size of vectors.

```
val rec length : ∀a:o, list⟨a⟩ ⇒ nat =
  fun l {
    case l {
      Nil    → Zero
      Cns[c] → S[length c.tl]
    }
  }

type vec⟨a:o, s:τ⟩ = {l ∈ list⟨a⟩ | length l ≡ s}
```

▶ Remark. In the definition of `vec`, the second parameter must have sort τ (the sort of terms) and not ι (the sort of values). Indeed, it is often required to work with vectors whose sizes are of the form `add n m` (see the definition of the `app` function below).

▶ Remark. There is no constraint on the type of `s` in the definition of `vec`. This means that it is possible to consider the type of vectors of size `true` for example, but it will be empty since the `length` function only returns natural numbers. One of the main advantages of this approach is that it is compatible with subtyping.

Let us stress that vectors can always be used as lists, independently of their size. The type of vectors is a subtype of the type of lists, as shown by the following function.

```
val vec_to_list : ∀a:o, ∀s:τ, vec⟨a,s⟩ ⇒ list⟨a⟩ =
  fun x { x }
```

Note that we will never need to use the function `vec_to_list` to turn a vector into a list. A vector can be seen as a list directly, without relying on any form of coercion.

We will now define a concatenation function app on vectors. It produces a vector whose length is the sum of the lengths of its two arguments. Note that we are first required to define the length_total function for a technical reason that will be explained in Section 5.[15]

```
val rec length_total : ∀a:ο, ∀l∈list⟨a⟩, ∃v:ι, v ≡ length l =
  fun l {
    case l {
      Nil    → {}
      Cns[c] → length_total c.tl
    }
  }

val rec app : ∀a:ο, ∀m n:ι, vec⟨a, m⟩ ⇒ vec⟨a, n⟩ ⇒ vec⟨a, add m n⟩ =
  fun l1 l2 {
    case l1 {
      Nil    → l2
      Cns[c] → length_total c.tl; Cns[{hd = c.hd; tl = app c.tl l2}]
    }
  }
```

Thanks to the Curry-style nature of our system, the sizes of the argument vectors do not need to be provided as arguments. This may be surprising for readers that are used to manipulating equivalent types in Agda or Coq, for example.

▶ Remark. In PML$_2$, the proof mechanism can also be used to eliminate unreachable code. Indeed, if an equational contradiction is triggered only by learning equations along the way, then the code in that branch cannot be accessed during evaluation. In this case, a special value ✂ (to be pronounced "scissors") can be used. Note that reachability information would be particularly useful to efficiently compile PML$_2$ programs down to assembly code.

4 Programs extracted from classical proofs

We will now consider an example of a program that can only be written in a classical setting (i.e., with control operators). We are going to define a function on streams of natural numbers called extract, that extracts a substream of odd numbers or a substream of even numbers from its input. This will prove that such a substream exists for all steams of natural numbers.[16] First, we need to define odd and even numbers using our set type syntax.

```
val rec is_odd : nat ⇒ bool =
  fun n {
    case n {
      Zero → false
      S[m] →
        case m {
          Zero → true
          S[p] → is_odd p
        }
    }
  }
```

[15] We have good hopes of simplifying this particular point in future work, for example by automatically obtaining length_total from the definition of length as they have a similar structure.

[16] Intuitively, we will have shown that every stream of natural numbers contains either infinitely many odd numbers or infinitely many even numbers (and possibly both).

```
type odd  = {v∈nat | is_odd v ≡ true }
type even = {v∈nat | is_odd v ≡ false}
```

As for the `length` function of the previous section, we will need to show that the `is_odd` function is total for a technical reason (see Section 5 for more details). Intuitively, this will allow us to reason by cases on the oddness (or evenness) of a given number of the input stream. Indeed, the totality of `is_odd` implies that this function always produces a result value, and hence that we can pattern match on its result.

```
val rec odd_total : ∀n∈nat, ∃v:ι, is_odd n ≡ v =
  fun n {
    case n {
      Zero → {}
      S[m] →
        case m {
          Zero → {}
          S[p] → odd_total p
        }
    }
  }
```

We also need to define the type of streams, together with a related type corresponding to streams with an explicit size annotation (or ordinal) `s`. Intuitively, this size annotation indicates the number of elements that are available in the stream (see Section 5 for more details on sized-types).

```
type corec stream⟨a⟩ = {} ⇒ {hd : a; tl : stream}
type sized_stream⟨s,a⟩ = ν_s stream, {} ⇒ {hd : a; tl : stream}
```

We can now define the `extract_aux` function, that will be used to define `extract` on the next page. Note that it relies on `abort`, which logically amounts to the *ex falso quodlibet* principle. Size annotations are also required on the type of `extract_aux`, for our type-checking algorithm to prove its termination.

```
val abort : ∀y, (∀x,x) ⇒ y = fun x { x }

val rec extract_aux : ∀a b,
    neg⟨sized_stream⟨a,even⟩⟩ ⇒
    neg⟨sized_stream⟨b,odd ⟩⟩ ⇒ neg⟨stream⟨nat⟩⟩ =
  fun fe fo s {
    let {hd ; tl} = s {};
    use odd_total hd;
    if is_odd hd {
      fo (fun _ {
        {hd = hd; tl = save oc {
          abort (extract_aux fe (fun x { restore oc x }) tl)}}
      })
    } else {
      fe (fun _ {
        {hd = hd; tl = save ec {
          abort (extract_aux (fun x { restore ec x }) fo tl)}}
      })
    }
  }
```

Intuitively, the `extract_aux` function looks at the head of its third argument (a stream of natural numbers), and depending on whether this number is odd or even, the function calls one of its first two arguments. They can be understood as partially constructed stream of even or odd numbers, in the form of continuations.[17] The read number is then added to this stream, and a recursive call is made to continue the construction.

▶ Remark. It may seem surprising that our prototype implementation is able to establish the termination of `extract_aux` as an element is added to one of two streams at each call. Moreover, this example does not satisfy the usually required semi-continuity condition [1]. It is here accepted because our termination test depends more finely on the structure of programs than previous approaches [19].

The `extract` function can then be defined as follows, to complete the construction. The function starts by saving two continuations, corresponding to the constructors `InL` and `InR` of the return type, and then calls `extract_aux` on the input stream.

```
val extract : stream⟨nat⟩ ⇒ either⟨stream⟨even⟩, stream⟨odd⟩⟩ =
  fun s {
    save a {
      InL[save ec { restore a InR[save oc {
        abort (extract_aux (fun x { restore ec x})
          (fun x { restore oc x }) s)
      } ] } ]
    }
  }
```

The very fact that we can write `extract` proves that it is possible to extract a stream of odd numbers or a stream of even numbers from any stream of natural numbers.

Of course, it is only possible to observe a finite prefix of a stream using a terminating program. As a consequence, we may want to consider a finite version of `extract`, whose result is a vector of a given size `n` instead of a stream.

```
val rec prefix : ∀a, ∀n∈nat, stream⟨a⟩ ⇒ vec⟨a,n⟩ =
  fun n s {
    case n {
      Zero → Nil
      S[k] → let {hd ; tl} = s {};
             Cns[{hd ; tl = prefix k tl}]
    }
  }

val finite_extract : ∀n∈nat,
    stream⟨nat⟩ ⇒ either⟨vec⟨even,n⟩, vec⟨odd,n⟩⟩ =
  fun n s {
    case extract s {
      InL[s] → InL[prefix n s]
      InR[s] → InR[prefix n s]
    }
  }
```

[17] Logical negation is intuitively used to type continuations represented in the form of a function. A continuation of type `neg⟨a⟩` can thus be called with a value of type `a` in any context since it yields a logical contradiction (or an element of type `∀x,x`).

▶ Remark. It is possible to give an equivalent definition of `finite_extract` in an intuitionistic setting (i.e., without using a control operator). Indeed, at most `2 × n` elements of the input stream need to be considered to construct the result.

▶ Remark. Of course, the type of `extract` or `finite_extract` does not directly imply that the result of these functions is a substream of their input. It is nonetheless easy to convince oneself that this is indeed the case, and we could certainly prove it in PML$_2$ with some effort.

To conclude this section, we will consider the result returned by `extract` (or rather `finite_extract`) on two particular streams. The former will be the stream of all natural numbers, which can be defined as follows, and is called `naturals`.

```
val rec naturals_from : nat ⇒ stream⟨nat⟩ =
  fun n _ {
    {hd = n; tl = naturals_from S[n]}
  }

val naturals : stream⟨nat⟩ = naturals_from Zero
```

The latter will be a stream of ones, prefixed by three zeroes. It can be defined as follows, and is called `three_zeroes_then_ones`.

```
val rec ones : stream⟨nat⟩ =
  fun _ { {hd = S[Zero]; tl = ones} }

val three_zeroes_then_ones : stream⟨nat⟩ =
  fun _ { {hd = Zero; tl =
    fun _ { {hd = Zero; tl =
      fun _ { {hd = Zero; tl = ones} }} }} }
```

The results of `finite_extract` on `naturals` and `three_zeroes_then_ones` may be displayed using the following printing functions.

```
val rec print_nat : nat ⇒ {} =
  fun n {
    case n {
      Zero → print "0"
      S[k] → print "S"; print_nat k
    }
  }

val rec print_list : ∀a, (a ⇒ {}) ⇒ list⟨a⟩ ⇒ {} =
  fun pelt l {
    case l {
      Nil           → print "\n"
      Cns[{hd;tl}] → pelt hd; print " "; print_list pelt tl
    }
  }

val print_res : either⟨list⟨nat⟩, list⟨nat⟩⟩ ⇒ {} =
  fun e {
    case e {
      InL[l] → print "  InL "; print_list print_nat l
      InR[l] → print "  InR "; print_list print_nat l
    }
  }
```

▶ Remark. Although the above is boiler-plate code, it is provided so that the examples in this document are completely self-contained, and can be type-checked and evaluated by PML$_2$ without any modification.

We have now all the components that are required to run some tests, and to display prefixes of the streams produced by the `extract` function. We will display, for each example, prefixes of increasing size. We will thus rely on the following `test` function, taking as input a stream of natural numbers, and showing the result of applying the `finite_extract` function on this stream with various prefix lengths (from zero to four).

```
val test : stream⟨nat⟩ ⇒ {} =
  fun s {
    print_res (finite_extract Zero s);
    print_res (finite_extract S[Zero] s);
    print_res (finite_extract S[S[Zero]] s);
    print_res (finite_extract S[S[S[Zero]]] s);
    print_res (finite_extract S[S[S[S[Zero]]]] s)
  }
```

Let us first consider the output that is produced by applying the above `test` function to `three_zeroes_then_ones`, which contains three zeroes followed by infinitely many ones.

```
InL
InL 0
InL 0 0
InR S0 S0 S0
InR S0 S0 S0 S0
```

As one should expect, the computation of the smallest prefixes yields a list of even numbers. However, if more elements of the input stream are read, the `extract` function eventually backtracks and produces a list of odd numbers instead. Indeed, the input stream only contains three even numbers.

Note that one could expect the fourth line of the output to show a list of even numbers with three zeroes. The produced result is due to the definition of `extract`, which looks ahead one element further than strictly necessary in the input stream. It would be possible to avoid doing so, but the function would be even more complex than it already is. Note that this also has consequences on the result obtained by running `test` on `naturals`.

```
InL
InL 0
InL 0 SS0
InL 0 SS0 SSSS0
InL 0 SS0 SSSS0 SSSSSS0
```

Here, one would expect the prefixes to alternate between lists of even numbers, and lists of odd numbers. Indeed, the stream of all the natural numbers both contain an infinite sub-stream of even numbers, and an infinite sub-stream of odd numbers.

5 Termination and internal totality proofs

We will now look more deeply into the relation between proofs and termination checking in PML$_2$. Technically, the termination of PML$_2$ programs (and thus of PML$_2$ proofs) is established using circular proof techniques introduced in joint work with Christophe Raffalli [19] and adapted in the author's thesis [17]. The idea is to type recursive programs, or more precisely the fixed-point combinator used by PML$_2$, using a simple unfolding rule. In other words, instances of the fixed-point construction of the language are typed assuming that they can be typed, thus leading to a circular structure. Of course, proofs constructed in this way may be invalid (i.e., not well-founded). To rule out such invalid proofs, a test based on the size-change principle [15] is used. When it is able to show that the structure of a proof is indeed well-founded, termination then follows from a standard semantic proof by realizability.[18]

5.1 Termination and consistency

As mentioned in the introduction, practical functional programming languages like OCaml or Haskell cannot be used to prove mathematical formulas, since their type system is not consistent when seen as a logic. More precisely, the “empty type” is inhabited by a simple looping program in these systems. Any formula can thus be proved through the *ex falso quodlibet* principle, as demonstrated by the following piece of Haskell code.[19]

```
type Empty = forall a. a

bad :: Empty
bad = bad

ex_falso :: Empty → a
ex_falso e = e
```

A similar example can also be given in OCaml, but a slightly more complex definition is required for `bad`.[20] This is due to the call-by-value evaluation strategy of the language, which restricts the use of the `let rec` construct to the definition of functions.

```
type empty = { any : 'a.'a }

let bad : empty =
  let rec bad_aux : unit → empty = fun () → bad_aux () in
  bad_aux ()

let ex_falso : type a. empty → a =
  fun e → e.any
```

Of course, a similar example can be written in PML$_2$. This is why the current implementation requires every program (not only proofs) to pass the termination check.

As PML$_2$ can be used to prove program equivalences, the inconsistency that would be introduced by possible non-termination would allow proving any program equivalence by inhabiting the corresponding type. Moreover, a non-terminating program would allow invalid program equivalences to be added to the equational context. The following invalid program (first given in Section 3) gives an example of such a scenario.

```
// val rec add_n_Zero_loop : ∀n∈nat, add n Zero ≡ n =
//   fun n {
//     add_n_Zero_loop n
//   }
```

[18] In this case, adequacy can still be proved by well-founded induction on the structure of the typing proof.
[19] Note that the `Rank2Types` (or `RankNTypes`) extension is required for the definition of `Empty`.
[20] Note that `empty` is encoded using a polymorphic record field.

Here, the recursive call `add_n_Zero_loop n` brings into the equational context the equivalence `add n Zero` $\equiv$ `n`, which exactly corresponds to the goal of the proof.

▶ Remark. The example `add_n_Zero_loop` must be rejected because the underlying program (and hence proof structure) is non terminating (and hence not well-founded). The management of equivalences being correct by construction, incorrect equations can only be proved in a contradictory equational context. In the example, a faulty equation is learned when the non-well-founded recursive call is made.

5.2 Sized types

The termination checking technology used by PML$_2$ is based on a notion of size that is attached to typing judgments. In fact, inductive and coinductive types are annotated using an ordinal size indicating the number of times their definition can be unfolded (this is usual in the context of sized types [1, 13, 30, 19]). Inductive or coinductive types, such as lists or streams, can then be seen as sized types annotated by a large enough (limit) ordinal.[21]

In practice, the implementation of PML$_2$ introduces sizes automatically when typing recursive functions. This means that it replaces (some) inductive and coinductive types, which carry a limit ordinal, with universal quantification over all possible ordinals. Note that it is only possible to do so when the obtained type is more general that the one that was given by the user. To enforce this invariant, we only introduce quantification on inductive types in negative position, and on coinductive types in positive position. In practice, this heuristic works well on simple functions, but the user is sometimes required to annotate the functions with explicit quantifications for termination to be established. Moreover, manual annotation may lead to more precise types, leading to more examples passing the termination check. For example, the following `map` function is accepted by the implementation.

```
val rec map : ∀a b, (a ⇒ b) ⇒ list⟨a⟩ ⇒ list⟨b⟩ =
  fun fn l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = fn c.hd; tl = map fn c.tl}]
    }
  }
```

However, if the user writes a complex recursive function containing a recursive call through the `map` function, it will not be possible to establish its termination (despite the fact that `map` does not change the size of the list it is applied to). To solve this problem, the user may rather use a more precise sized type, which is a subtype of the former type.

```
type slist⟨s:κ, a:ο⟩ = μ_s slist, [Nil ; Cns of {hd : a; tl : slist}]

val rec map : ∀s, ∀a b, (a ⇒ b) ⇒ slist⟨s,a⟩ ⇒ slist⟨s,b⟩ =
  fun fn l {
    case l {
      Nil    → Nil
      Cns[c] → Cns[{hd = fn c.hd; tl = map fn c.tl}]
    }
  }
```

[21] In practice, ω is sufficient for most of the usual data types, but this is not true in general. Nonetheless, there exists an ordinal that is large enough for all the inductive and coinductive types to converge [19].

▶ Remark. The type `slist⟨s,a⟩` should not be confused with the type of vectors `vec⟨a,s⟩` defined in Section 3.4. Although they are both subtypes of regular lists, the former carries an ordinal s (of sort κ) that can be used by the termination checker to establish size relations, while the latter contains a term (of sort τ) corresponding to the size of the list (as computed by the `length` function) which cannot be used by the termination checker. Note however that these two types could be easily combined.

▶ Remark. The above type of `map` only enforces that the output list is at most as long as the input list. For instance, we could give the same type to a function taking the same two arguments and always returning an empty list.

A similar scheme can be applied to *insertion sort* for example, but not for *quick sort*, as discussed in previous work [19]. Indeed, a richer language of ordinals would be required to express the fact that the partition function preserves the number of elements of its input.[22]

5.3 Proof by equivalence to a terminating function

Although the current implementation of PML$_2$ checks the termination of all programs (not only proofs), it is possible to use the specific features of the system to write termination proofs. Indeed, the equivalence relation on which the system relies can be used to substitute one term with another, provided that they are equivalent. This means that if we want to establish the termination of a program that does not pass the termination check directly, then we can instead establish the termination of any equivalent program. We will here consider the example of the well-known *McCarthy 91* function, whose termination cannot be established by most of the existing termination criteria (if not all).

```
include lib.nat
include lib.nat_proofs

def mccarthy91_hard =
  fix fun mccarthy91 n {
    if gt n u100 {
      minus n u10
    } else {
      mccarthy91 (mccarthy91 (add n u11))
    }
  }

// val mccarthy91 : nat ⇒ nat =
//   mccarthy91_hard
```

Note that here, the value `mccarthy91_hard` is not defined as a usual, type checked and termination checked value, but as a value object (using the `def` keyword). This means that this function can be manipulated as an object of the logic, but not evaluated directly. Note also that we rely on some functions (and constants) defined in the standard library of PML$_2$. The `minus` function computes the difference, and the `gt` function tests whether its first argument is strictly greater than its second argument.

Although PML$_2$ is not able to prove the termination of the commented version of `mccarthy91`, we can give the following alternative (but equivalent) definition.

[22] It is nonetheless possible to show that quick sort is size-preserving using a PML$_2$ proof.

```
val mccarthy91_easy : nat ⇒ nat =
  fun n {
    if gt n u100 {
      minus n u10
    } else {
      u91
    }
  }
```

This second definition passes our termination check (it is not even recursive), but it does not really correspond to the traditional definition of the *McCarthy 91 function*, which is a shame. We can nonetheless write a PML$_2$ proof showing that these definitions are (pointwise) equivalent, which will then allow us to replace one with the other. To do so, we first need to show that `mccarthy91_hard n` has value `u91` for all numbers that are not greater than `u100`.

```
val hard_aux: ∀n∈nat, gt n u100 ≡ false ⇒ mccarthy91_hard n ≡ u91 =
  fun n eq {
    {- ... -} // Can be done by enumerating the domain.
  }
```

We do not give the full proof for lack of space, but it can be easily completed since the domain of quantification is finite. One simply needs to explore the domain by pattern matching on `n`, obtaining a trivial proof for all numbers less or equal to `u100`. In the case of numbers greater that `u100`, the presence of an additional successor produces a contradiction with the hypothesis `gt n u100 ≡ false` which allows the enumeration to remain finite.

▶ Remark. This brute-force approach, although it could be easily automated, yields a proof that it rather inefficient. A better solution would be to write a proof by “induction”, which is what one would do on paper.

Using the `hard_aux` lemma, we can then show that the two implementations of the *McCarthy 91 function* produce the same result on every natural number as follows.

```
val hard_is_easy : ∀n∈nat, mccarthy91_easy n ≡ mccarthy91_hard n =
  fun n {
    use gt_total n u100;
    if gt n u100 {
      deduce mccarthy91_easy n ≡ minus n u10;
      deduce mccarthy91_hard n ≡ minus n u10;
      qed
    } else {
      deduce mccarthy91_easy n ≡ u91;
      show mccarthy91_hard n ≡ u91 using hard_aux n {};
      qed
    }
  }
```

The proof is straightforward[23] since the two implementations have the same structure, and they share the same “then” branch. In the case of the “else” branch, `hard_aux` can be used to conclude. We can then type check and prove the termination of the original version of the *McCarthy 91 function* as follows.

[23] None of the `deduce` annotations are necessary, they are only provided for clarity. The `gt_total` lemma is defined in the standard library, more detail about its purpose will be given in the next section.

```
val mccarthy91 : nat ⇒ nat =
  fun n {
    check mccarthy91_easy n  // Term used for type-checking.
      for mccarthy91_hard n  // Actual term used in the definition.
      because hard_is_easy n // Proof that they are equal.
    // The above really is "mccarthy91_hard n" (up to erasure).
  }
```

The annotation used in the definition of `mccarthy91` instructs the type-checker to substitute `mccarthy_hard n` with `mccarthy_easy n` in the construction of the typing proof. This is only possible because these two terms are equivalent (when `n` has type `nat`), as witnessed by `hard_is_easy n`. However, the term used for the computation will indeed be `mccarthy_hard n` after the annotations are erased.

▶ Remark. As all the types of PML$_2$ are closed under equivalence, it is always possible to replace a term by another equivalent term. This technique can not only be used for proving termination of functions such as `mccarthy91`, but also for typing terms that would not be typable otherwise (but that are, for example, more efficient).

▶ Remark. Note that we did not prove `mccarthy_hard` ≡ `mccarthy_easy`, which may not even be true. Indeed, equivalence considers these two terms as untyped, and it is very well possible that they can be distinguished by a certain evaluation context.[24] A simpler example arises when comparing different implementations of the identity function on natural numbers: we have `(fun n { case n { Zero ⇒ n | S[_] ⇒ n } }) k ≡ (fun n { n }) k` for all `k` in `nat`, but `fun n { case n { Zero ⇒ n | S[_] ⇒ n } } ≡ fun n { n }` is false. These two functions can be distinguished using the argument `false`, which yields a pattern matching failure on the former, while the latter successfully returns `false`.

5.4 Internal totality proofs

In this last section, we will give more explanations about the so-called "totality proofs" that are currently required in PML$_2$. A function is said to be *total* if it computes some value, when applied to any value of its domain. In PML$_2$, the totality of functions can be expressed inside the system using an existential quantification. We can thus write internal totality proofs such as the following.

```
val rec add_total : ∀n m∈nat, ∃v:ι, add n m ≡ v =
  fun n m {
    case n {
      Zero → qed
      S[k] → use add_total k m; qed
    }
  }
```

▶ Remark. Note that the value that is obtained by applying the function is not relevant here, nor is its type. We could however modify the definition of `add_total` to make sure that an element of type `nat` is returned. In this case, we could even use `add_total` as `add`.

[24] In PML$_2$, the equivalence `t ≡ u` being provable implies that `t` and `u` are observationally equivalent, which means that they have the same "observable behaviour" in every possible evaluation context. Note that we only observe termination, versus divergence or runtime error [17, 16].

The reason why totality proofs are required in the system is strongly related to the call-by-value evaluation strategy of the language. Indeed, in call-by-value, a function can only be applied when all of its arguments are values. More precisely, it only makes sense to reduce a β-redex if the term in argument position is a syntactic value. To understand where the notion of totality is really required, let us consider the following proof example showing the associativity of addition.

```
val rec add_assoc : ∀m n p∈nat, add m (add n p) ≡ add (add m n) p =
  fun m n p {
    use add_total n p;
    case m {
      Zero → qed
      S[k] → use add_assoc k n p; use add_total k n; qed
    }
  }
```

Ignoring the first call to `add_total`, the proof starts by a case analysis on variable `m`. Let us consider the `Zero` case, which already illustrates very well the necessity for the totality proof. In this branch, the automatic decision procedure learns the equation `m ≡ Zero`. As a consequence, the goal simplifies to `add Zero (add n p) ≡ add (add Zero n) p`, and even as `add Zero (add n p) ≡ add n p` since both `Zero` and `n` are values (the function can thus be applied). However, the left-hand side of the equation cannot reduce further because `add n p` is not a value. We can then only proceed using the totality proof produced by `add_total n p`, which gives us a value `v` such that `add n p ≡ v`. As a consequence, this allows us to obtain `add Zero (add n p) ≡ add n p` as follows.

$$\texttt{add Zero (add n p)} \quad\equiv\quad \texttt{add Zero v} \quad\equiv\quad \texttt{v} \quad\equiv\quad \texttt{add n p}$$

▶ Remark. It is clear that the totality proof corresponding to a given function has a similar structure as the definition of the function itself. We may thus hope that totality proofs can be generated and called automatically, at least in most cases.

6 Future work

The current implementation of PML$_2$ already allows for several convincing examples, some of which cannot be expressed in other systems. They include, for example, the `extract` function of Section 4 or the `mccarthy91` function of Section 5.3. However, theoretical work and implementation work remain to be done for the language to become fully practical, both as a programming language and as a proof assistant.

6.1 Mixing termination and non-termination

As mentioned earlier, termination checking is only necessary for PML$_2$ programs that are considered as proofs. In the theory, proving that a program terminates amounts to showing that its typing derivation has a well-founded circular structure. In this case, a standard semantic proof can be used to prove normalisation [19], the essential point being that the adequacy of the type system can be established by induction on the circular structure of proofs[25], provided that they are well-founded.

[25] Circularity is introduced by the typing rule for the fixed-point combinator.

Alternatively, it is possible to type programs with standard (non-circular) typing proofs,[26] to the expense of losing normalisation since our termination criterion works by analysing the circular structure of proofs. Note that lack of termination checking implies the loss of soundness, but type-safety is nonetheless preserved. As it is hard to automatically prove the termination of programs, it is clear that a user will not want to be restricted to programs that can be proved terminating. For this purpose, it is important to allow arbitrary (type-safe) programs to be written, if only to prove them terminating later (examples of such programs can be found in previous work [27]).

As programs that can be proven terminating can be typed in both ways, it is natural to consider a way of mixing the two approaches in the theory. This has actually already been implemented in a particular branch of our implementation (called *totality*) [18]. The corresponding extension of the theory has also been checked informally.

6.2 Other forms of effects, mutation

One of the distinguishing features of PML$_2$ is the possibility for programs to manipulate their own continuation (or evaluation context). This is achieved using a construct similar to Scheme's *call/cc*, or rather Michel Parigot's μ-abstraction [25], which triggers a form of effect. As shown in Section 2.1, it can be used to realize theorems which only hold classically, by extracting a program from their proofs [26].

Although PML$_2$ is the first proof system based on a programming language with effects and a classical realizability model [17], one may argue that control structures only have a limited interest for writing practical ML programs.[27] Other forms of effects however, for example input/output directives or mutable cells, are essential to ML programmers. Although it should be relatively easy to extend the system with the former, the latter poses a real technical challenge. Indeed, it is not yet known how to account for mutation in a classical realizability model.

6.3 Subject reduction and strong safety

The theory of PML$_2$ is based on a realizability model, which has the major advantage of being flexible. More precisely, the adequacy lemma, which is the keystone of the development, only needs to be modified locally to encompass a new typing or subtyping rule. However, we have not yet proved any subject reduction result for the system, and thus we only have a weak form of type safety.

6.4 Extensible variants and records (better inference)

The current type-system of PML$_2$ requires a relatively small amount of type annotations (at least for programs). Nonetheless, the system relies on unification in several places, and it may happen that the system guesses the wrong types. This situation arises most often with variant and record types, for which some fields or constructors might be left out. This problem can be solved using extensible variant types and record types, but we will need to make sure that this does not pose any problem in the theory.

[26] This feature is not available in the current implementation, which only accepts terminating programs.
[27] They can however be used to encode a form of exception mechanism.

6.5 Support for mutually recursive function

In the current implementation, PML$_2$ lacks the possibility of defining mutually recursive functions. Although it is always possible to encode mutual recursion using additional parameters, this method does not perform very well when combined with our termination checking technology. We thus need to consider a different fixed-point instruction for our abstract machine, which does not seem to pose any theoretical problem. The idea is to replace the current fixed-point instruction with a term constructor $\varphi a.v$, binding the term variable a into the value v, and with the reduction rule $\varphi a.v \to v[a := \varphi a.v]$.[28] Mutual recursion can then be encoded using a value v that is a record containing several λ-abstractions, which can still be typed using a simple unfolding rule (as in previous work [19, 17]).

6.6 Certificates using proof traces for equivalences

For now, it is not possible to formally check the proofs produced by PML$_2$ in another system. Although the system already records the proof trees that are produced during type-checking, the decision procedure for program equivalence yet lacks the ability of producing a proof trace. However, there is no theoretical evidence that it would not be possible for the decision procedure to record enough information for an external prover (for example Coq [23] or Dedukti [31]) to check the proofs produced by PML$_2$.

7 Similar systems

To conclude this paper, we will compare PML$_2$ to other proof systems and languages that can be used to formalise and prove program properties, or that rely on similar principles.

7.1 Dependent types in ML

To our knowledge, the combination of call-by-value evaluation, side-effects and dependent products has never been achieved before. At least not for a dependent product fully compatible with effects and call-by-value. For example, the Aura language [14] forbids dependency on terms that are not values in dependent applications. Similarly, the $F^\star$ language [32] relies on (partial) let-normal forms to enforce values in argument position. Daniel Licata and Robert Harper have defined a notion of positively dependent types [20] which only allow dependency over strictly positive types. Finally, in languages like ATS and DML [36, 37], dependencies are limited to a specific index language.

7.2 Tools based on intuitionistic type theory

The most actively developed proof assistants following the Curry-Howard correspondence are Agda and Coq [24, 23]. The former is based on Martin-Löf's dependent type theory and the latter on Coquand and Huet's calculus of constructions [7, 22]. These two constructive theories provide dependent types, which allow the definition of very expressive specifications. Contrary to PML$_2$, Coq and Agda do not directly give a computational interpretation to classical logic. Classical reasoning can only be done through a negative translation or with the definition of axioms such as the law of the excluded middle. In particular, these two

[28] Term variables should not be confused with value variables (or λ-variables). In particular, the former can be substituted with any term, while the latter can only be substituted with values.

languages are not effectful. However, they are logically consistent, which means that they only accept terminating programs. As termination checking is a difficult (and undecidable) problem, many terminating programs are rejected. Although this is not a problem for formalizing mathematics, this makes programming tedious. In PML_2, only proofs really need to be shown terminating, and it is in any case possible to reason about non-terminating and even untyped programs as they can be manipulated as objects in types.

7.3 NuPrl and refinement types

The NuPrl system [6] has many similarities with PML_2 on the theoretical side, although it is inconsistent with classical logic. NuPrl accommodates an observational equivalence relation similar to ours (Howe's *squiggle* relation [12]), which is partially reflected in the syntax of the system. Being based on a Kleene-style realizability model, NuPrl can also be used to reason about untyped terms. Another major difference between PML_2 and NuPrl is that the latter is based on refinement types, which means that it does not have an automatic way of building typing derivations for programs. Indeed, typing derivations are built interactively using a specific interface, and the user must say what typing rule should be applied first.

7.4 Partially consistent languages

The TRELLYS project [3] aims at providing a language in which a consistent core interacts with type-safe dependently typed programming with general recursion. Although the language is call-by-value and effectful, it suffers from value restriction like Aura [14]. The value restriction does not appear explicitly but is encoded into a well-formedness judgement appearing as the premise of the typing rule for application. Apart from value restriction, the main difference between the language of the TRELLYS project and ours resides in the calculus itself. Their calculus is Church-style (or explicitly typed) while ours is Curry-style (or implicitly typed). In particular, their terms and types are defined simultaneously, while our type system is constructed on top of an untyped calculus.

7.5 Systems aimed at program verification in ML

Several systems have been proposed for verifying ML programs. ProPre [21] relies on a notion of *algorithms*, corresponding to equational specifications of programs. It is used in conjunction with a type system based on intuitionistic logic. Although it is possible to use classical logic to prove that a program meets its specification, the underlying programming language is not effectful. Similarly, the PAF! system [2] implements a logic supporting proofs of programs, but it is restricted to a purely functional subset of ML. Another approach for reasoning about purely functional ML programs is given in the work of Yann Regis-Gianas [29], where Hoare logic is used to specify program properties. Finally, it is also possible to reason about ML programs (including effectful ones) by compiling them down to higher-order formulas [4, 5], which can then be manipulated using an external prover such as Coq [23]. In this case, the user is required to master at least two languages, contrary to our system in which programming and proving take place in a uniform framework.

References

1 Andreas Abel. Semi-Continuous Sized Types and Termination. *Logical Methods in Computer Science*, 4(2), 2008.

2 Sylvain Baro. *Conception et implémentation d'un système d'aide à la spécification et à la preuve de programmes ML*. PhD thesis, Paris Diderot University, France, 2003.

3 Chris Casinghino, Vilhelm Sjöberg, and Stephanie Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, pages 33–46. ACM, 2014.

4 Arthur Charguéraud. Program verification through characteristic formulae. In *ICFP*, pages 321–332. ACM, 2010.

5 Arthur Charguéraud. Characteristic formulae for the verification of imperative programs. In *ICFP*, pages 418–430. ACM, 2011.

6 Robert L. Constable, Stuart F. Allen, Mark Bromley, Rance Cleaveland, J. F. Cremer, R. W. Harper, Douglas J. Howe, Todd B. Knoblock, N. P. Mendler, Prakash Panangaden, James T. Sasaki, and Scott F. Smith. *Implementing mathematics with the Nuprl proof development system.* Prentice Hall, 1986.

7 Thierry Coquand and Gérard P. Huet. The Calculus of Constructions. *Inf. Comput.*, 76(2/3):95–120, 1988.

8 Jacques Garrigue. Programming with polymorphic variants. In *ML Workshop*, 1998.

9 Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris 7, 1972.

10 Timothy Griffin. A Formulae-as-Types Notion of Control. In *POPL*, pages 47–58. ACM Press, 1990.

11 Robert Harper and Mark Lillibridge. ML with callcc is unsound. Posted to the SML mailing list, 1991. URL: `http://www.seas.upenn.edu/~sweirich/types/archive/1991/msg00034.html`.

12 Douglas J. Howe. Equality In Lazy Computation Systems. In *LICS*, pages 198–203. IEEE Computer Society, 1989.

13 John Hughes, Lars Pareto, and Amr Sabry. Proving the Correctness of Reactive Systems Using Sized Types. In *POPL*, pages 410–423. ACM Press, 1996.

14 Limin Jia, Jeffrey A. Vaughan, Karl Mazurak, Jianzhou Zhao, Luke Zarko, Joseph Schorr, and Steve Zdancewic. AURA: a programming language for authorization and audit. In *ICFP*, pages 27–38. ACM, 2008.

15 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92. ACM, 2001.

16 Rodolphe Lepigre. A Classical Realizability Model for a Semantical Value Restriction. In *ESOP*, volume 9632 of *Lecture Notes in Computer Science*, pages 476–502. Springer, 2016.

17 Rodolphe Lepigre. *Semantics and Implementation of an Extension of ML for Proving Programs. (Sémantique et Implantation d'une Extension de ML pour la Preuve de Programmes)*. PhD thesis, Grenoble Alpes University, France, 2017.

18 Rodolphe Lepigre and Christophe Raffalli. Implementation of the PML$_2$ language. The source code is available on GitHub at `https://github.com/rlepigre/pml`, 2017.

19 Rodolphe Lepigre and Christophe Raffalli. Practical Subtyping for Curry-Style Languages. *to appear in ACM Trans. Program. Lang. Syst.*, 2018. Draft and joined implementation available at `https://rlepigre.github.io/subml/`.

20 Daniel R. Licata and Robert Harper. Positively dependent types. In *PLPV*, pages 3–14. ACM, 2009.

21 Pascal Manoury, Michel Parigot, and Marianne Simonot. ProPre A Programming Language with Proofs. In *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 484–486. Springer, 1992.

22 Per Martin-Löf. Constructive mathematics and computer programming. In L. Jonathan Cohen, Jerzy Łoś, Helmut Pfeiffer, and Klaus-Peter Podewski, editors, *Logic, Methodology and Philosophy of Science VI*, volume 104 of *Studies in Logic and the Foundations of Mathematics*, pages 153–175. North-Holland, 198.

23 The Coq development team. *The Coq proof assistant reference manual.* LogiCal Project, 2018. Version 8.8.0. URL: `http://coq.inria.fr`.

24 Ulf Norell. Dependently Typed Programming in Agda. In *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.

25 Michel Parigot. Lambda-Mu-Calculus: An Algorithmic Interpretation of Classical Natural Deduction. In *LPAR*, volume 624 of *Lecture Notes in Computer Science*, pages 190–201. Springer, 1992.

26 Christophe Raffalli. Getting results from programs extracted from classical proofs. *Theor. Comput. Sci.*, 323(1-3):49–70, 2004.

27 Christophe Raffalli. PML: a new proof assistant. Prototype implementation available at `http://lama.univ-savoie.fr/~raffalli/pml`, talk at the TYPES workshop, 2007.

28 John C. Reynolds. Towards a Theory of Type Structure. In *Programming Symposium, Proceedings Colloque sur la Programmation*, pages 408–423. Springer-Verlag, 1974.

29 Yann Régis-Gianas. *From types to logical assertions : automatic or assisted proofs of property about functional programs. (Des types aux assertions logiques : preuve automatique ou assistée de propriétés sur les programmes fonctionnels).* PhD thesis, Paris Diderot University, France, 2007.

30 Jorge Luis Sacchini. Type-Based Productivity of Stream Definitions in the Calculus of Constructions. In *LICS*, pages 233–242. IEEE Computer Society, 2013.

31 Ronan Saillard. *Typechecking in the lambda-Pi-Calculus Modulo : Theory and Practice. (Vérification de typage pour le lambda-Pi-Calcul Modulo : théorie et pratique).* PhD thesis, Mines ParisTech, France, 2015.

32 Nikhil Swamy, Juan Chen, Cédric Fournet, Pierre-Yves Strub, Karthikeyan Bhargavan, and Jean Yang. Secure distributed programming with value-dependent types. *J. Funct. Program.*, 23(4):402–451, 2013.

33 Philip Wadler. Call-by-value is dual to call-by-name. *SIGPLAN Notices*, 38(9):189–201, 2003.

34 Andrew K. Wright. Simple Imperative Polymorphism. *Lisp and Symbolic Computation*, 8(4):343–355, 1995.

35 Andrew K. Wright and Matthias Felleisen. A Syntactic Approach to Type Soundness. *Inf. Comput.*, 115(1):38–94, 1994.

36 Hongwei Xi. Applied Type System: Extended Abstract. In *TYPES*, volume 3085 of *Lecture Notes in Computer Science*, pages 394–408. Springer, 2003.

37 Hongwei Xi and Frank Pfenning. Dependent Types in Practical Programming. In *POPL*, pages 214–227. ACM, 1999.

Formalized Proof Systems for Propositional Logic

Julius Michaelis
Technische Universität München, Germany
j.michaelis@tum.de
https://orcid.org/0000-0003-4626-3722

Tobias Nipkow
Technische Universität München, Germany
nipkow@in.tum.de
https://orcid.org/0000-0003-0730-515X

Abstract

We have formalized a range of proof systems for classical propositional logic (sequent calculus, natural deduction, Hilbert systems, resolution) in Isabelle/HOL and have proved the most important meta-theoretic results about semantics and proofs: compactness, soundness, completeness, translations between proof systems, cut-elimination, interpolation and model existence.

2012 ACM Subject Classification Theory of computation → Proof theory

Keywords and phrases formalization of logic, proof systems, sequent calculus, natural deduction, resolution

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.5

Acknowledgements We thank Peter Lammich for a first version of function *sc* and Jasmin C. Blanchette for valuable comments.

1 Introduction

This paper presents a unified formalization in Isabelle/HOL [29] of the classical proof systems for classical propositional logic (sequent calculus, natural deduction, Hilbert systems, resolution) and their meta-theory: compactness, soundness, completeness, translations between proof systems, cut-elimination, interpolation and model existence. The motivation is the desire to develop (in the long run) a counterpart to the emerging *Encyclopaedia of Proof Systems* [51], including all formalized meta-theory. Our work is also part of IsaFoL [3], a collection of formalizations of logics in Isabelle. A second motivation comes from teaching logic. Most textbooks cover only specific proof systems and specific results proved in a specific manner, and it is hard to modify the setup. That is why we cover multiple proof systems with their interconnections. In particular we give multiple proofs of some key results, e.g. three different completeness proofs. The longer term goal is a formalized logic textbook in the style of *Concrete Semantics* [28] but with a web of proof systems and multiple proofs of key results such that one can select one's own subset of the material. It is also possible to obtain verified executable code from such formalizations, e.g. our toy sequent calculus and resolution provers and our Craig interpolator.

Our paper differs from each of the related papers below in at least one of the following aspects: we do not reason about deductive systems alone but also about semantics; we do not consider a specific calculus but a collection of canonical calculi; we do not prove a specific result but most if not all the results covered in the propositional section of an introductory logic course. That is, we provide a unique, unified library of definitions and proofs. The formalization of the translations between all the proof systems appears to be new.

The complete development is available online [26]. It consists of 5000 lines of Isabelle text that translate into more than 100 A4 pages of definitions and proofs. For readability the notation in this paper differs in places from the Isabelle text.

1.1 Related Work

Formalization of logic was probably started by Shankar's proof of Gödel's first incompleteness theorem [42]. Harrison formalized basic first-order model theory [16]. There are many other formalizations of results about propositional and first-order logics (e.g. cut elimination [33, 49], interpolation [9], completeness [36, 8, 39, 21, 6], incompleteness [31], resolution [37, 38], quantifier elimination [27]) and temporal logic (e.g. [11]). Berghofer [1] formalized part of [13] but this was never published. There is also significant work on formalized SAT solvers [50, 25, 43, 24, 30, 5]. Harrison [17] started to verify a theorem prover in itself, which was later extended [23]. The above publications rely on general purpose theorem provers. Alternatively, a number of metalogical frameworks for reasoning about deductive systems have been implemented (e.g. [32, 34, 14, 35]) and proof theoretic results were proved in them, e.g. cut elimination in intuitonistic and classical logic [33], cut admissibility for a number of propositional linear logics [10], and translations between various calculi for minimal propositional logic of implication (in the Abella [14] library).

2 Isabelle Notation

The logic of Isabelle/HOL conforms largely to everyday mathematical notation. This section summarizes non-standard notation.

The type of truth values is called *bool*. The function space is denoted by $\Rightarrow$. Type variables are denoted by $'a$, $'b$, etc. The notation $t :: \tau$ means that term t has type τ. Type constructors follow postfix syntax, e.g. $'a$ *set* is the type of sets of elements of type $'a$. Lists over type $'a$, type $'a$ *list*, come with the empty list $[\,]$ and the infix constructor "$\cdot$", the infix append operator @. Function *set* converts a list into a set.

Type $'a$ *multiset* is the type of finite multisets over type $'a$. The empty multiset is $\emptyset_\#$, comprehension is written $\{...\}_\#$, multiset union is denoted by $+$, insertion by a comma, difference by $-$ and membership by $\in_\#$.

Horizontal lines in rules are displayed versions of the implication $\longrightarrow$.

Many of our theorems are proved by induction on the structure of formulas or derivations. If we do not say anything about how the induction cases are dealt with, this means that Isabelle can come up with proofs for them automatically (often with the help of Sledgehammer [2]).

3 Formulas and Their Semantics

A formula (which is simply a term of a recursive data type) is either an atom $A_0, A_1, \ldots$[1], or $\bot$, or constructed from other formulas using the connectives $\mathbf{\neg}$, $\mathbf{\wedge}$, $\mathbf{\vee}$ and $\mathbf{\to}$; note the bold font to distinguish them from $\neg$, $\wedge$, $\vee$ and $\longrightarrow$ in the metalogic. The variables F, G and H always stand for formulas. The function *atoms* returns the set of all atom indices in a formula (or similar object).

The semantics of a formula F is defined via a *valuation* or *assignment* $\mathcal{A} :: nat \Rightarrow bool$ (a function from atom indices to truth values) and a recursively defined operator $\mathcal{A} \models F$. The

[1] We use natural numbers to index atoms, but any countably infinite set would suffice.

$$\frac{}{A_k,\, \Gamma \Rightarrow A_k,\, \Delta}\ \textsc{Ax} \qquad \frac{}{\bot,\, \Gamma \Rightarrow \Delta}\ \textsc{BotL}$$

$$\frac{\Gamma \Rightarrow F,\, \Delta}{\neg\, F,\, \Gamma \Rightarrow \Delta}\ \textsc{NotL} \qquad \frac{F,\, \Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \neg\, F,\, \Delta}\ \textsc{NotR}$$

$$\frac{F,\, G,\, \Gamma \Rightarrow \Delta}{F \wedge G,\, \Gamma \Rightarrow \Delta}\ \textsc{AndL} \qquad \frac{\Gamma \Rightarrow F,\, \Delta \qquad \Gamma \Rightarrow G,\, \Delta}{\Gamma \Rightarrow F \wedge G,\, \Delta}\ \textsc{AndR}$$

$$\frac{F,\, \Gamma \Rightarrow \Delta \qquad G,\, \Gamma \Rightarrow \Delta}{F \vee G,\, \Gamma \Rightarrow \Delta}\ \textsc{OrL} \qquad \frac{\Gamma \Rightarrow F,\, G,\, \Delta}{\Gamma \Rightarrow F \vee G,\, \Delta}\ \textsc{OrR}$$

$$\frac{\Gamma \Rightarrow F,\, \Delta \qquad G,\, \Gamma \Rightarrow \Delta}{F \rightarrow G,\, \Gamma \Rightarrow \Delta}\ \textsc{ImpL} \qquad \frac{F,\, \Gamma \Rightarrow G,\, \Delta}{\Gamma \Rightarrow F \rightarrow G,\, \Delta}\ \textsc{ImpR}$$

Figure 1 Rules of sequent calculus.

$\models$ operator canonically maps the constructors of the formulas to their equivalent boolean operators in the metalogic, e.g.

$$(\mathcal{A} \models F \rightarrow G) = (\mathcal{A} \models F \longrightarrow \mathcal{A} \models G).$$

Atoms are evaluated by looking up their index in $\mathcal{A}$: $\mathcal{A} \models A_k = \mathcal{A}\ k$. Assignments are total functions and thus suitable for any formula.

4 Proof Systems

4.1 Sequent Calculus

We formalize a Sequent Calculus (SC) inspired by Troelstra and Schwichtenberg [48]. But whereas they define derivation trees inductively, we merely define derivable sequents, where sequents are pairs of multisets. That is, we formalize an inductively defined predicate $\Gamma \Rightarrow \Delta$ where Γ and Δ are multisets. Hence $\Gamma \Rightarrow \Delta$ means that the sequent is derivable. We begin with the system that [48] refers to as **G3c**: its rules, adjusted for our definition of formulas, are shown in Figure 1.

In the literature one finds three types of sequents: lists, multisets and sets. After some experiments with all three representations we found that multisets were most convenient, although we did not conduct an in-depth study. However, we did formalize Gentzen's original system (approximately **G2c** in [48]) based on lists (with all the structural rules) and proved the equivalence with the multiset-based system SC from Figure 1. From now on we consider the latter system only.

Many of the proofs in [48] are by induction over the depth of a derivation tree. In order to be able to follow this style we defined a second, ternary predicate written $\Gamma \Rightarrow_n \Delta$, where n is the depth of the derivation, and proved $(\exists\, n.\ \Gamma \Rightarrow_n \Delta) \longleftrightarrow \Gamma \Rightarrow \Delta$. A canonical definition of depth would have Ax and BotL start with 0 (or 1) and use the maximum in rules with two premises. Instead, we require both antecedents to have the same "depth", e.g.:

$$\frac{\Gamma \Rightarrow_n F,\, \Delta \qquad G,\, \Gamma \Rightarrow_n \Delta}{F \rightarrow G,\, \Gamma \Rightarrow_{n\,+\,1} \Delta} \qquad \frac{F,\, \Gamma \Rightarrow_n G,\, \Delta}{\Gamma \Rightarrow_{n\,+\,1} F \rightarrow G,\, \Delta} \qquad \frac{}{A_k,\, \Gamma \Rightarrow_{n\,+\,1} A_k,\, \Delta}$$

We found this modification to be more suitable for Isabelle's automated reasoning tools.

While this allows the proofs in the next section to follow [48], we additinally found proofs that do not require depth arguments. We present only those.

4.1.1 Cut Admissibility

We prove weakening, inversion, contraction and cut as admissible rules in the meta-logic. The proofs of the eight inversion rules can be partly automated, but they are still quite long. One major issue is exchange: given a sequent of the form $F,\ G \rightarrow H,\ \Gamma \Rightarrow \Delta$, we need to exchange F and $G \rightarrow H$ before we can apply ImpL. The problem is that Isabelle's unification algorithm does not know about multiset laws. Although we do not need to apply an explicit exchange rule of SC, the proof still needs to apply the multiset equation $F,\ G,\ \Gamma = G,\ F,\ \Gamma$. Luckily, ordered rewriting (which Isabelle's simplifier supports) with this equation sorts sequents into a canonical order, thus automating the application of this equation in many cases. Another reason for our proofs to be long is that we formalized derivable sequents. When a proof in the literature makes a case distinction on whether a formula is principal, we can only reproduce that by making a case distinction on which rule was applied last. That requires discharging ten instead of two cases.

The proof of the admissibility of the cut rule is by induction on the cut formula as in [48]. The induction steps are entirely taken care of automatically. For the base cases, we need two auxiliary lemmas:

$$\frac{\Gamma \Rightarrow \Delta}{\Gamma \Rightarrow \Delta - \{\bot\}_\#} \qquad \frac{A_k,\ \Gamma \Rightarrow \Delta \qquad \Gamma \Rightarrow A_k,\ \Delta}{\Gamma \Rightarrow \Delta}$$

They are proved by induction on the derivations of $\Gamma \Rightarrow \Delta$ and $A_k,\ \Gamma \Rightarrow \Delta$. While the second lemma is just the cut rule for atoms, it is easier to show than the cut rule itself since we know that the cut formula cannot be principal in the induction steps of its proof.

The proof of the two contraction rules proceeds in exactly the same manner, unlike the proof by induction on the derivation tree depth in [48]. The proof by induction on the depth does not require the two auxiliary lemmas in the base cases but is slightly more unwieldy in Isabelle/HOL.

4.1.2 Soundness and Completeness

We define the semantics of a sequent as follows:

$$\mathcal{A} \models \Gamma \Rightarrow \Delta \overset{\text{def}}{=} (\forall F {\in_\#} \Gamma.\ \mathcal{A} \models F) \longrightarrow (\exists F {\in_\#} \Delta.\ \mathcal{A} \models F).$$

Note that $\_ \models \_ \Rightarrow \_$ is a ternary operator. Validity is defined as usual:

$$\models \Gamma \Rightarrow \Delta \overset{\text{def}}{=} \forall \mathcal{A}.\ \mathcal{A} \models \Gamma \Rightarrow \Delta$$

Soundness ($\Gamma \Rightarrow \Delta \longrightarrow\ \models \Gamma \Rightarrow \Delta$) is proved by induction on $\Gamma \Rightarrow \Delta$. The proof of completeness

$$\models \Gamma \Rightarrow \Delta \longrightarrow \Gamma \Rightarrow \Delta \tag{1}$$

is more involved and conceptually similar to Gallier's [15, §3.4.6]. Gallier presents a search procedure that constructs derivation trees. Once the procedure terminates (for propositional logic, it always does), the set of open leaves in the derivation tree corresponds to the set of counterexamples. Our proof search procedure *sc* dispenses with derivation trees and works on lists of formulas and atoms. Replacing multisets with lists is necessary to formulate *sc* as a simple structurally recursive function. In a call *sc* $\Gamma\ A\ \Delta\ B$, the parameters Γ and Δ are lists of formulas and (ignoring A and B) we are trying to prove *mset* $\Gamma \Rightarrow$ *mset* Δ, where *mset* converts a list into a multiset. In each step, the first formula in Γ or Δ is decomposed according to a rule of SC. Atoms (more precisely, their indices) are moved into A and B. If

$$
\begin{array}{lcl}
sc\ [\,]\ A\ [\,]\ B & = & \text{if } set\ A \cap set\ B = \emptyset \text{ then } \{(A,\ B)\} \text{ else } \emptyset \\
sc\ (A_k \cdot \Gamma)\ A\ \Delta\ B & = & sc\ \Gamma\ (k \cdot A)\ \Delta\ B \\
sc\ \Gamma\ A\ (A_k \cdot \Delta)\ B & = & sc\ \Gamma\ A\ \Delta\ (k \cdot B) \\
sc\ (\bot \cdot \Gamma)\ A\ \Delta\ B & = & \emptyset \\
sc\ \Gamma\ A\ (\bot \cdot \Delta)\ B & = & sc\ \Gamma\ A\ \Delta\ B \\
sc\ (\neg\ F \cdot \Gamma)\ A\ \Delta\ B & = & sc\ \Gamma\ A\ (F \cdot \Delta)\ B \\
sc\ \Gamma\ A\ (\neg\ F \cdot \Delta)\ B & = & sc\ (F \cdot \Gamma)\ A\ \Delta\ B \\
sc\ (F \wedge G \cdot \Gamma)\ A\ \Delta\ B & = & sc\ (F \cdot G \cdot \Gamma)\ A\ \Delta\ B \\
sc\ \Gamma\ A\ (F \wedge G \cdot \Delta)\ B & = & sc\ \Gamma\ A\ (F \cdot \Delta)\ B \cup sc\ \Gamma\ A\ (G \cdot \Delta)\ B \\
sc\ (F \vee G \cdot \Gamma)\ A\ \Delta\ B & = & sc\ (F \cdot \Gamma)\ A\ \Delta\ B \cup sc\ (G \cdot \Gamma)\ A\ \Delta\ B \\
sc\ \Gamma\ A\ (F \vee G \cdot \Delta)\ B & = & sc\ \Gamma\ A\ (F \cdot G \cdot \Delta)\ B \\
sc\ (F \to G \cdot \Gamma)\ A\ \Delta\ B & = & sc\ \Gamma\ A\ (F \cdot \Delta)\ B \cup sc\ (G \cdot \Gamma)\ A\ \Delta\ B \\
sc\ \Gamma\ A\ (F \to G \cdot \Delta)\ B & = & sc\ (F \cdot \Gamma)\ A\ (G \cdot \Delta)\ B
\end{array}
$$

Figure 2 Search procedure *sc*.

$\Gamma = \Delta = [\,]$ and $set\ A \cap set\ B = \emptyset$, then a counterexample has been found. The defining equations for *sc* are shown in Figure 2.

From a pair $(A,\ B)$ returned by *sc*, a countermodel can be constructed: map all elements of A to *True* and all elements of B to *False*. The following propositions are not hard to prove:

- The search procedure always terminates in at most n steps, where $n = 2 * \sum(map\ size\ (\Gamma\ @\ \Delta)) + length\ (\Gamma\ @\ \Delta)$ and the *size* of a formula is the number of connectives in it. For the same n we proved:
- $sc\ \Gamma\ [\,]\ \Delta\ [\,] = \emptyset \longrightarrow mset\ \Gamma \Rightarrow_n mset\ \Delta$
- $(A,\ B) \in sc\ \Gamma\ [\,]\ \Delta\ [\,] \longrightarrow \lambda k.\ k \in set\ A \not\models mset\ \Gamma \Rightarrow mset\ \Delta$

Generalizations of the latter two facts (with variables for the empty lists) follow by induction on the computation of *sc*. The completeness of SC follows easily. Note that in the final proposition the countermodel is based only on A because disjointness of A and B implies that $\lambda k.\ k \in set\ A$ maps elements in B to *False*.

4.2 Natural Deduction

We have chosen a presentation of Natural Deduction (ND) with explicit contexts.[2] A context Γ is a set of formulas. The set of ND rules is shown in Figure 3. Explicit contexts require rule Ax to get started. Streamlined proofs with explicit contexts require a weakening rule which is proved by induction on $\Gamma \vdash_N F$:

$$\frac{\Gamma \vdash_N F \qquad \Gamma \subseteq \Gamma'}{\Gamma' \vdash_N F}$$

4.2.1 Soundness and Completeness

The semantic meaning of $\Gamma \vdash_N F$ is best described by entailment[3]:

[2] This is indispensable for a positive inductive definition: without contexts, ImpI becomes $(\vdash_N F \longrightarrow \vdash_N G) \longrightarrow \vdash_N F \to G$ where the inductive predicate occurs in a negative position $(\vdash_N F)$.

[3] We chose a separate symbol $\Vdash$ for entailment because further overloading of $\models$ necessitates type annotations for disambiguation in too many places.

$$\frac{F \in \Gamma}{\Gamma \vdash_N F}\,\textsc{Ax} \qquad \frac{\{\neg\, F\} \cup \Gamma \vdash_N \bot}{\Gamma \vdash_N F}\,\mathrm{CC}$$

$$\frac{\Gamma \vdash_N \neg\, F \qquad \Gamma \vdash_N F}{\Gamma \vdash_N \bot}\,\textsc{NotE} \qquad \frac{\{F\} \cup \Gamma \vdash_N \bot}{\Gamma \vdash_N \neg\, F}\,\textsc{NotI}$$

$$\frac{\Gamma \vdash_N F \wedge G}{\Gamma \vdash_N F}\,\textsc{AndE}_1 \qquad \frac{\Gamma \vdash_N F \wedge G}{\Gamma \vdash_N G}\,\textsc{AndE}_2 \qquad \frac{\Gamma \vdash_N F \qquad \Gamma \vdash_N G}{\Gamma \vdash_N F \wedge G}\,\textsc{AndI}$$

$$\frac{\Gamma \vdash_N F}{\Gamma \vdash_N F \vee G}\,\textsc{OrI}_1 \qquad \frac{\Gamma \vdash_N G}{\Gamma \vdash_N F \vee G}\,\textsc{OrI}_2$$

$$\frac{\Gamma \vdash_N F \vee G \qquad \{F\} \cup \Gamma \vdash_N H \qquad \{G\} \cup \Gamma \vdash_N H}{\Gamma \vdash_N H}\,\textsc{OrE}$$

$$\frac{\{F\} \cup \Gamma \vdash_N G}{\Gamma \vdash_N F \to G}\,\textsc{ImpI} \qquad \frac{\Gamma \vdash_N F \to G \qquad \Gamma \vdash_N F}{\Gamma \vdash_N G}\,\textsc{ImpE}$$

Figure 3 Rules of natural deduction.

$$\Gamma \Vdash F \overset{\text{def}}{=} \forall \mathcal{A}.\ (\forall\, G{\in}\Gamma.\ \mathcal{A} \models G) \longrightarrow \mathcal{A} \models F.$$

Soundness, $\Gamma \vdash_N F \longrightarrow \Gamma \Vdash F$, is shown by induction on $\Gamma \vdash_N F$.

For showing completeness, we follow an approach presented, for example, by Huth and Ryan [20]: show that ND can "simulate truth tables". The translation from assignments to formulas is captured by the notation $F^{\mathcal{A}}$ defined by

$$F^{\mathcal{A}} \overset{\text{def}}{=} \textsf{if } \mathcal{A} \models F \textsf{ then } F \textsf{ else } \neg\, F$$

First we show two lemmas:

$$\mathit{atoms}\ F \subseteq Z \longrightarrow \{A_k{}^{\mathcal{A}} \mid k \in Z\} \vdash_N F^{\mathcal{A}}$$
$$(\forall \mathcal{A}.\ \{A_k{}^{\mathcal{A}} \mid k \in \{l\} \cup Z\} \vdash_N F) \longrightarrow (\forall \mathcal{A}.\ \{A_k{}^{\mathcal{A}} \mid k \in Z\} \vdash_N F)$$

The first one is proved by induction on F, the second one requires a few lines of careful manual reasoning involving the derived rule

$$\frac{\{F\} \cup \Gamma \vdash_N H \qquad \{\neg\, F\} \cup \Gamma \vdash_N H}{\Gamma \vdash_N H}$$

A limited completeness theorem $\models F \longrightarrow \emptyset \vdash_N F$ follows by downward induction on the size of *atoms* F (removing atom by atom). The general completeness theorem requires compactness and is postponed to Section 6.

4.3 Hilbert Systems

We consider Hilbert systems (HS) based on the axioms and rules shown in Figure 4. Axioms 5 to 10 correspond to those originally postulated by Hilbert [19]. Axioms 1 and 2 are merely a more compact way of expressing Hilbert's first four axioms. The axioms from 11 onward needed to be adjusted since Hilbert does not use formulas with $\bot$.

The deduction theorem is proved in the standard inductive fashion:

$$\frac{\{F\} \cup \textsc{Axs}_0 \cup \Gamma \vdash_H G}{\textsc{Axs}_0 \cup \Gamma \vdash_H F \to G}$$

Here $\textsc{Axs}_0$ denotes the set of axioms 1 and 2 only. In Section 5 we relate HS to ND and SC, thus obtaining a syntactic proof of soundness and completeness. A semantic proof of completeness is given in Section 9.

1. $F \to G \to F$
2. $(F \to G \to H) \to (F \to G) \to F \to H$
5. $(F \wedge G) \to F$
6. $(F \wedge G) \to G$
7. $F \to G \to F \wedge G$
8. $F \to F \vee G$
9. $G \to F \vee G$
10. $(F \to H) \to (G \to H) \to (F \vee G) \to H$
11. $(F \to \bot) \to \neg F$
12. $\neg F \to F \to \bot$
13. $(\neg F \to \bot) \to F$

$$\frac{F \in \Gamma}{\Gamma \vdash_H F}\,\text{Ax}$$

$$\frac{\Gamma \vdash_H F \qquad \Gamma \vdash_H F \to G}{\Gamma \vdash_H G}\,\text{MP}$$

Figure 4 Axioms and rules of Hilbert systems.

5 Translating Between Proof Systems

This section presents results relating derivations in SC, ND and HS. The key results are:

$$\Gamma \vdash_N F \longrightarrow \text{Axs}_1 \cup \Gamma \vdash_H F \tag{2}$$
$$\text{Axs}_1 \cup \{F \mid F \in_\# \Gamma\} \vdash_H G \longrightarrow \Gamma \Rightarrow \{G\}_\# \tag{3}$$
$$\Gamma \Rightarrow \Delta \longrightarrow \{F \mid F \in_\# \Gamma\} \cup \{\neg F \mid F \in_\# \Delta\} \vdash_N \bot \tag{4}$$

where Axs_1 is the full set of axioms from Figure 4. All of these proofs are by induction on the rules of the system in the premise, simulating each rule in the system of the conclusion.

For the proof of (2) the deduction theorem is required wherever ND modifies its context Γ. The only case that needs some manual attention is that of simulating ORE, simply because of its slightly higher complexity.

For (3), we need to show that all Axs_1 axioms are derivable in SC, which can be done by blindly applying matching SC rules. To simulate MP, admissibility of the cut rule is necessary.

For (4), we need a set of rules that execute the reasoning of SC in the context Γ of ND, prefixing a $\neg$ to formulas that would appear in Δ, for example:

$$\frac{\{\neg F, \neg G\} \cup \Gamma \vdash_N \bot}{\{\neg (G \vee F)\} \cup \Gamma \vdash_N \bot} \qquad \frac{\{\neg F\} \cup \Gamma \vdash_N \bot \qquad \{G\} \cup \Gamma \vdash_N \bot}{\{F \to G\} \cup \Gamma \vdash_N \bot}$$

The proofs of these rules are constructed automatically.

Overall we obtain that the following three propositions are equivalent:

$$\text{Axs}_1 \vdash_H F \qquad \emptyset \vdash_N F \qquad \emptyset_\# \Rightarrow \{F\}_\#.$$

6 Compactness

The compactness theorem states that *satisfiability* and *finite satisfiability* of a set of formulas coincide:

$$\begin{aligned} &\textit{sat } S &&\overset{\text{def}}{=} \exists \mathcal{A}.\ \forall F {\in} S.\ \mathcal{A} \models F \\ &\textit{fin\_sat } S &&\overset{\text{def}}{=} \forall s {\subseteq} S.\ \textit{finite } s \longrightarrow \textit{sat } s \end{aligned}$$

We follow Enderton's proof [12] which is based on an enumeration of all formulas. There is an Isabelle library theory that can automatically derive countability of the type of formulas and define a surjective function from a natural number n to a formula F_n. This enables us to define the saturation of a set of formulas:

$$\frac{C \in S}{S \vdash_R C} \qquad \frac{S \vdash_R C \qquad S \vdash_R D \qquad P_k \in C \qquad N_k \in D}{S \vdash (C - \{k^+\}) \cup (D - \{k^-\})}$$

Figure 5 Rules of Resolution.

$$\begin{aligned}
&\textit{saturate } S\ 0 = S\\
&\textit{saturate } S\ (n+1) =\\
&(\mathsf{let}\ S' = \textit{saturate } S\ n;\ S_t = \{F_n\} \cup S';\ S_f = \{\neg\ F_n\} \cup S'\\
&\ \mathsf{in\ if}\ \textit{fin\_sat}\ S_f\ \mathsf{then}\ S_f\ \mathsf{else}\ S_t)\\
&\textit{Saturate } S \stackrel{\text{def}}{=} \textstyle\bigcup_n \textit{saturate } S\ n
\end{aligned}$$

Compactness follows after proving the following lemmas:

$$\begin{aligned}
&S \subseteq \textit{Saturate } S\\
&\textit{fin\_sat } S \longrightarrow F \in \textit{Saturate } S \longleftrightarrow \neg\ F \notin \textit{Saturate } S\\
&\textit{fin\_sat } S \longrightarrow \textit{fin\_sat } (\textit{Saturate } S)\\
&\textit{fin\_sat } S \longrightarrow (\lambda k.\ A_k \in \textit{Saturate } S) \models F \longleftrightarrow F \in \textit{Saturate } S
\end{aligned}$$

The proofs are standard.

As an example application of compactness, we show a general completeness theorem for ND: $\Gamma \models F \longrightarrow \Gamma \vdash_N F$. We have proved this both via the completeness result from Section 4.2, compactness and weakening, and via compactness, the SC completeness result from Section 4.1 and (4) from Section 5.

7 Resolution

We have dedicated a separate section to resolution since it is quite different from the other proof systems. To begin with, it uses a different type of formulas: CNFs. A CNF is a set of clauses, where a clause is a set of positive (P_k) or negative (N_k) literals. We use S, T for CNFs, C, D, E for clauses and L for literals. We write the empty clause as $\square$. The semantics of CNFs (and literals) is defined in a similar fashion to that of formulas, and we will use the same $\models$ for it here:

$$\begin{aligned}
&\mathcal{A} \models S \stackrel{\text{def}}{=} \forall\, C{\in}S.\ \exists\, L{\in}C.\ \mathcal{A} \models L,\\
&\mathcal{A} \models P_k = \mathcal{A}\ k, \quad \mathcal{A} \models N_k = (\neg\ \mathcal{A}\ k).
\end{aligned}$$

To convert a formula into CNF, we first convert it into NNF ($\neg$ is applied only to atoms) by pushing $\neg$ downward and eliminating $\rightarrow$ in the usual fashion. To convert a formula in NNF into CNF, we define another recursive function: *cnf* :: *formula* $\Rightarrow$ *literal set set*. Its only interesting equation is

$$\textit{cnf}\ (F \vee G) = \{C \cup D \mid C \in \textit{cnf } F \wedge D \in \textit{cnf } G\}.$$

The semantic correctness of these transformations ($\mathcal{A} \models \textit{cnf}\ (\textit{nnf } F) \longleftrightarrow \mathcal{A} \models F$) is trivial.

Our formalization of resolution as an inductively defined predicate requires two rules, as shown in Figure 5. A resolution refutation of S is a derivation $S \vdash_R \square$. The following two weakening rules will come in handy:

$$\frac{S \vdash_R C}{T \cup S \vdash_R C}\,(5) \qquad \frac{S \cup T \vdash_R C}{\exists\, D{\subseteq}\{L\}.\ \{\{L\} \cup E \mid E \in S\} \cup T \vdash_R D \cup C}\,(6)$$

A weaker version of (6) is presented by Gallier [15, §4.3.4] roughly as (we leave out any formal notation): take a resolution refutation graph and insert an additional atom into some of the start nodes' clauses, then apply the same resolution steps as in the original graph. We have to generalize this from assuming a refutation to an arbitrary resolution $S \cup T \vdash_R C$ to show the claim by induction on the derivation.

7.1 Soundness and Completeness

Soundness ($S \vdash_R \Box \longrightarrow \forall \mathcal{A}.\ \mathcal{A} \not\models S$) is an easy corollary of the following lemma which is proved by induction on $S \vdash_R C$:

$$S \vdash_R C \wedge \mathcal{A} \models S \longrightarrow \mathcal{A} \models \{C\}$$

We give two proofs of completeness. The first one follows Schöning [40]. Since the proof is an induction on the set of atoms in the CNF, we need the following definitions to manipulate that set:

$$\begin{aligned} k^v &\overset{\text{def}}{=} \textsf{case } v \textsf{ of } \mathit{True} \Rightarrow P_k \mid \mathit{False} \Rightarrow N_k \\ S[v/k] &\overset{\text{def}}{=} \{C - \{k^{\neg v}\} \mid C \in S \wedge k^v \notin C\} \end{aligned}$$

The notational similarity with a substitution is deliberate, but beware that there is no new formula or clause that gets substituted in. Instead, the CNF is modified as if that atom index k had been set to the value v:

$$\mathcal{A} \models S[v/k] \longleftrightarrow \mathcal{A}(k := v) \models S$$

We follow Schöning and show

$$(\forall \mathcal{A}.\ \mathcal{A} \not\models S) \wedge \mathit{finite}\ (\mathit{atoms}\ S) \longrightarrow S \vdash_R \Box$$

by induction on $\mathit{atoms}\ S$. The base case is trivial since S can only contain the empty clause. In the step case we conclude that if $\forall \mathcal{A}.\ \mathcal{A} \not\models S$, then also $\forall \mathcal{A}.\ \mathcal{A} \not\models S[v/k]$, and hence $S[v/k] \vdash_R \Box$ for any v and some $k \in \mathit{atoms}\ S$. For a slick formal proof of the final $S \vdash_R \Box$, it is necessary to find a suitable lemma that relates a CNF $S[v/k]$ back to the original CNF S:

$$\{\textsf{if } \{k^{\neg v}\} \cup C \in S \textsf{ then } \{k^{\neg v}\} \cup C \textsf{ else } C \mid C \in S[v/k]\} \subseteq S$$

After splitting the set comprehension into two separate sets of clauses based on whether $\{k^{\neg v}\} \cup C \in S$, we can use the two weakening lemmas (5,6) and resolution to finish the proof.

7.2 A Translation Between SC and Resolution

We provide a translation between SC proofs and resolution refutations. This yields a second soundness and completeness argument. The idea is to translate from an SC proof $\Gamma \Rightarrow \emptyset_\#$ into a resolution refutation $\Gamma' \vdash_R \Box$ where Γ' is a CNF corresponding to Γ. SC in the form presented in Section 4.1 is not very suitable for this kind of reasoning. Instead, we follow an idea by Gallier [15] and introduce *LSC* (shown in Figure 6), our own variant of Schütte's one-sided calculus K_1 [41]. We prove

$$\Gamma \Rightarrow \emptyset_\# \longleftrightarrow \Gamma \Rightarrow_L$$

To go from LSC to resolution we require a special normal form of formulas. The formula should be a conjunction of disjunctions of literals, but with a twist: the conjunctions can be arbitrarily nested but the disjunctions have to be nested to one side (we chose the right side). This format is captured by the predicate *is_cnf* on formulas. With that, we can show

$$\frac{}{\neg\ A_k,\ A_k,\ \Gamma \Rightarrow_L}\ \textsc{Ax} \qquad \frac{}{\bot,\ \Gamma \Rightarrow_L}\ \textsc{BotL}$$

$$\frac{F,\ \Gamma \Rightarrow_L}{\neg\ (\neg\ F),\ \Gamma \Rightarrow_L}$$

$$\frac{F,\ G,\ \Gamma \Rightarrow_L}{F \wedge G,\ \Gamma \Rightarrow_L}\ \textsc{AndL} \qquad \frac{\neg\ F,\ \Gamma \Rightarrow_L \qquad \neg\ G,\ \Gamma \Rightarrow_L}{\neg\ (F \wedge G),\ \Gamma \Rightarrow_L}$$

$$\frac{F,\ \Gamma \Rightarrow_L \qquad G,\ \Gamma \Rightarrow_L}{F \vee G,\ \Gamma \Rightarrow_L}\ \textsc{OrL} \qquad \frac{\neg\ F,\ \neg\ G,\ \Gamma \Rightarrow_L}{\neg\ (F \vee G),\ \Gamma \Rightarrow_L}$$

$$\frac{\neg\ F,\ \Gamma \Rightarrow_L \qquad G,\ \Gamma \Rightarrow_L}{F \rightarrow G,\ \Gamma \Rightarrow_L} \qquad \frac{F,\ \neg\ G,\ \Gamma \Rightarrow_L}{\neg\ (F \rightarrow G),\ \Gamma \Rightarrow_L}$$

Figure 6 Rules of LSC.

$$(\forall\, F {\in_\#} \Gamma.\ \mathit{is\_cnf}\ F) \wedge \Gamma \Rightarrow_L\ \longrightarrow \bigcup\{\mathit{cnf}\ F \mid F \in_\# \Gamma\} \vdash_R \Box$$

by induction on $\Gamma \Rightarrow_L$. The motivation for the special nesting of disjunctions is the OrL case. In order to apply (6) one of the disjuncts must be a literal that can match L in (6). The idea for this trick is borrowed from Gallier [15], but the details of the proofs differ (cf. [26]).

In a second step we can use the lemma above to prove our main result:

$$\{F\}_\# \Rightarrow \emptyset_\# \longrightarrow \mathit{cnf}\ (\mathit{nnf}\ F) \vdash_R \Box$$

Doing so requires a number of auxiliary lemmas with laborious proofs about converting deductions of $\Gamma \Rightarrow_L$ to deductions of $\Gamma' \Rightarrow_L$ where Γ' is a variant of Γ such that *is_cnf* holds for all elements of Γ'.

We have also shown $S \vdash_R \Box \longrightarrow \exists\, F.\ \mathit{cnf}\ (\mathit{nnf}\ F) \subseteq S \wedge \{F\}_\# \Rightarrow \emptyset_\#$. We do not go into this proof.

7.3 A Toy Resolution Prover

It is possible to generate code from executable Isabelle definitions; inductive definitions are interpreted in a Prolog-like manner. Although applicable to $\vdash_R$, it leads to the usual nontermination issue due to DFS. Thus we implement resolution by hand in two steps. First we define a function *res* that computes all resolvents of a clause set:

$$\begin{array}{l} \mathit{res}\ S \overset{\text{def}}{=} (\bigcup C_1 \in S.\ \bigcup C_2 \in S.\ \bigcup L_1 \in C_1.\ \bigcup L_2 \in C_2. \\ \quad \mathsf{case}\ (L_1,L_2)\ \mathsf{of} \\ \quad\ (P_i,N_j) \Rightarrow \mathit{if}\ i = j\ \mathit{then}\ \{(C_1 - \{P_i\}) \cup (C_2 - \{N_j\})\}\ \mathit{else}\ \emptyset \\ \quad \mid \_ \qquad\quad \Rightarrow \emptyset) \end{array}$$

Then we iterate *res* until no new clauses are generated:

$$\mathit{Res}\ S = (\mathsf{let}\ R = \mathit{res}\ S \cup S\ \mathsf{in}\ \mathsf{if}\ R = S\ \mathsf{then}\ \mathit{Some}\ S\ \mathsf{else}\ \mathit{Res}\ R)$$

The result is wrapped in **datatype** *'a option* = *None* | *Some 'a* to express if the computation diverged or produced a result a by returning *None* or *Some a*. For more details see [22, 7]. Because of Isabelle's automatic data refinement of sets by lists this is an executable function. We proved soundness and completeness wrt. $\vdash_R$ and termination:

$$\begin{array}{l} \mathit{Res}\ S = \mathit{Some}\ T \longrightarrow (C \in T) = S \vdash_R C \\ \mathit{finite}\ S \wedge (\forall\, C {\in} S.\ \mathit{finite}\ C) \longrightarrow \exists\, T.\ \mathit{Res}\ S = \mathit{Some}\ T \end{array}$$

Of course we can only show termination for finite sets of finite clauses. The termination proof relies on the fact that there is a finite bounding set, the set of all clauses that can be constructed from the atoms in S, because resolution does not introduce new atoms.

Of course *Res* is inefficient and only useful for demonstration purposes. Efficient variants would need much further refinement as in [4, 5].

8 Craig Interpolation

The interpolation theorem states: given an SC derivation $\Gamma_1 + \Gamma_2 \Rightarrow \Delta_1 + \Delta_2$, there exists an *interpolant* G s.t.

$$\Gamma_1 \Rightarrow G, \Delta_1 \text{ and } G, \Gamma_2 \Rightarrow \Delta_2 \text{ and}$$
$$\mathit{atoms}\ G \subseteq \mathit{atoms}\ (\Gamma_1 + \Delta_1) \text{ and } \mathit{atoms}\ G \subseteq \mathit{atoms}\ (\Gamma_2 + \Delta_2)$$

(A slightly less general version uses $\Gamma_2 = \Delta_1 = \emptyset_\#$.) We follow the proof by Troelstra and Schwichtenberg [48], which is by induction on the depth of the derivation. Instead of formalizing the split sequents used in the original proof, we use multiset unions. Troelstra and Schwichtenberg only provide the cases for Ax, BotL, ImpL and ImpR.

To avoid the other six cases, we rewrite formulas into the implicative fragment, i.e we define a transformation from a formula F to a formula $\overrightarrow{F}$:

$$\begin{array}{lcl@{\qquad\qquad}lcl}
\overrightarrow{A_k} & = & A_k & \overrightarrow{\bot} & = & \bot \\
\overrightarrow{F \to G} & = & \overrightarrow{F} \to \overrightarrow{G} & \overrightarrow{F \wedge G} & = & (\overrightarrow{F} \to \overrightarrow{G} \to \bot) \to \bot \\
\overrightarrow{\neg F} & = & \overrightarrow{F} \to \bot & \overrightarrow{F \vee G} & = & (\overrightarrow{F} \to \bot) \to \overrightarrow{G}
\end{array}$$

Showing that this transformation preserves the semantics is trivial. Since we do not want to introduce a semantic argument into our proof of interpolation, we also need to show that derivability in SC is preserved by the transformation:

$$\Gamma \Rightarrow \Delta \longleftrightarrow \overrightarrow{\Gamma} \Rightarrow \overrightarrow{\Delta}$$

We show this using one induction over the SC rules for each direction of the logical equivalence. In the direction from left to right, derivations can be transformed easily. The direction from right to left is technically more difficult: the induction will produce two cases where $\to$ was the topmost connective of the last principal formula, but the $\to$ could be the result of rewriting any of the four connectives $\neg$, $\wedge$, $\vee$ or $\to$. Appropriately splitting this into cases requires some manual effort. This means that the reduced set of connectives does not reduce the total proof effort for Craig interpolation dramatically, but we can also apply it to contraction and cut admissibility.

The interpolation theorem is proved by induction on $\Gamma \Rightarrow \Delta$ where $\Gamma = \Gamma_1 + \Gamma_2$ and $\Delta = \Delta_1 + \Delta_2$. In the proof, we can now assume that $\neg$, $\wedge$ and $\vee$ do not occur anywhere, making the corresponding six cases vacuous. The remaining four cases need to be split into a total of 10 subcases distinguishing in which of the multisets Γ_1, Δ_1, Γ_2, Δ_2 the principal formula occurred. In each of the subcases, we provide a witness for G. [4]. Some of the difficulty of the proof lies in finding a way to instantiate the induction hypothesis and then finding a possible interpolation formula for the given instantiation. Consider, for example, the ImpL case with $F \to H$ principal and $F \to H \in_\# \Gamma_1$. For brevity, we define $\Gamma_1{}'$ s.t. $\Gamma_1 = F \to H, \Gamma_1{}'$. Now, we have (at least) two choices: We can follow Troelstra and Schwichtenberg and instantiate the IHs to obtain interpolants G_F and G_H such that:

[4] In [48] one of the witnesses for Ax is $\bot$ where it should be $\bot \to \bot$.

$\Gamma_2 \Rightarrow G_F, \Delta_2$ and $G_F, {\Gamma_1}' \Rightarrow F, \Delta_1$ and
$H, {\Gamma_1}' \Rightarrow G_H, \Delta_1$ and $G_H, \Gamma_2 \Rightarrow \Delta_2$

The new interpolant is then $G_F \rightarrow G_H$. Note that this is not the only possible interpolant. Alternatively we can obtain a G_F that satisfies ${\Gamma_1}' \Rightarrow G_F, F, \Delta_1$ and $G_F, \Gamma_2 \Rightarrow \Delta_2$ and use $G_F \vee G_H$ as the interpolant.

While the above proof is fully constructive and provides witnesses for G in all cases, splitting the sequent leads to a lot of technical detail. The next subsection shows an easy semantic alternative.

8.1 Craig Interpolation via Substitution

It is also possible to show that if $\models F \rightarrow H$, then there exists a formula G s.t.

$\models F \rightarrow G$ and $\models G \rightarrow H$ and
$\mathit{atoms}\ G \subseteq \mathit{atoms}\ F \cap \mathit{atoms}\ H$

Doing so is interesting because there is a much easier proof with a different approach to constructing the interpolant G, e.g. as presented by Harrison [18]: provide a uniform recursive function that computes the interpolant:

```
$\mathit{interpolate}\ F\ H =$
($\mathsf{let}\ K = \mathit{atoms}\ F - \mathit{atoms}\ H$
 $\mathsf{in\ if}\ K = \emptyset\ \mathsf{then}\ F$
    $\mathsf{else\ let}\ k = \mathit{Min}\ K\ \mathsf{in}\ \mathit{interpolate}\ (F[(\bot \rightarrow \bot)/k] \vee F[\bot/k])\ H)$
```

where $F[G/k]$ substitutes G for A_k in F and *Min* is (arbitrarily) used to select the minimal index. The proof is by induction on the set $\mathit{atoms}\ F - \mathit{atoms}\ H$. The induction step follows using $\mathcal{A} \models F \longrightarrow \mathcal{A} \models F[(\bot \rightarrow \bot)/k] \vee F[\bot/k]$.

9 The Model Existence Theorem

So far we have invested a lot of work into constructive, or at least insightful proofs. This section shows a way to derive many of the results with much less effort: the Model Existence Theorem, as presented by Fitting [13] (originally by Smullyan [44, 45]). We begin by defining when a set of formulas is a Hintikka set:

$$\begin{aligned}
\mathit{Hintikka}\ S \overset{\text{def}}{=} {} & \bot \notin S \wedge (\forall k.\ A_k \in S \longrightarrow \neg\ A_k \in S \longrightarrow \mathit{False}) \wedge \\
& (\forall F\ G.\ F \wedge G \in S \longrightarrow F \in S \wedge G \in S) \wedge \\
& (\forall F\ G.\ F \vee G \in S \longrightarrow F \in S \vee G \in S) \wedge \\
& (\forall F\ G.\ F \rightarrow G \in S \longrightarrow \neg\ F \in S \vee G \in S) \wedge \\
& (\forall F.\ \neg\ (\neg\ F) \in S \longrightarrow F \in S) \wedge \\
& (\forall F\ G.\ \neg\ (F \wedge G) \in S \longrightarrow \neg\ F \in S \vee \neg\ G \in S) \wedge \\
& (\forall F\ G.\ \neg\ (F \vee G) \in S \longrightarrow \neg\ F \in S \wedge \neg\ G \in S) \wedge \\
& (\forall F\ G.\ \neg\ (F \rightarrow G) \in S \longrightarrow F \in S \wedge \neg\ G \in S)
\end{aligned}$$

Hintikka's lemma shows that Hintikka sets are satisfiable:

$\mathit{Hintikka}\ S \longrightarrow \mathit{sat}\ S$

We continue by defining when a set T of sets of formulas is a propositional consistency property:

$$\begin{aligned}
\mathit{pcp}\ T \ &\overset{\text{def}}{=}\ \forall S{\in}T.\ \bot \notin S \wedge (\forall k.\ A_k \in S \longrightarrow \neg\, A_k \in S \longrightarrow \mathit{False}) \wedge \\
&(\forall F\ G.\ F \wedge G \in S \longrightarrow \{F,\ G\} \cup S \in T) \wedge \\
&(\forall F\ G.\ F \vee G \in S \longrightarrow \{F\} \cup S \in T \vee \{G\} \cup S \in T) \wedge \\
&(\forall F\ G.\ F \rightarrow G \in S \longrightarrow \{\neg\, F\} \cup S \in T \vee \{G\} \cup S \in T) \wedge \\
&(\forall F.\ \neg\, (\neg\, F) \in S \longrightarrow \{F\} \cup S \in T) \wedge \\
&(\forall F\ G.\ \neg\, (F \wedge G) \in S \longrightarrow \{\neg\, F\} \cup S \in T \vee \{\neg\, G\} \cup S \in T) \wedge \\
&(\forall F\ G.\ \neg\, (F \vee G) \in S \longrightarrow \{\neg\, F,\ \neg\, G\} \cup S \in T) \wedge \\
&(\forall F\ G.\ \neg\, (F \rightarrow G) \in S \longrightarrow \{F,\ \neg\, G\} \cup S \in T)
\end{aligned}$$

Fitting's proof [13] of the Model Existence Theorem

$$\mathit{pcp}\ T \wedge S \in T \longrightarrow \mathit{sat}\ S$$

shows that every *pcp* can be extended to one where every member is the start of a $\subseteq$-chain (constructed in a similar fashion to the sequence in Section 6). The limit of this sequence is a Hintikka set, and by Hintikka's lemma satisfiable. The theorem follows. Our formalization of this proof matches Fitting's proof very closely, and we do not want to repeat it here. Instead, we present three examples where we applied the Model Existence Theorem:

- For SC, we show the lemma

  $$\mathit{pcp}\ \{\{F \mid F \in_{\#} \Gamma\} \mid \Gamma \not\Rightarrow \emptyset_{\#}\}.$$

  The proof of this is constructed automatically after dealing with the slight discrepancies between sets and multisets. Completeness (1) follows easily.
- We show compactness using the fact that

  $$\mathit{pcp}\ \{W \mid \mathit{fin\_sat}\ W\}.$$

- We show completeness of HS using

  $$\mathit{pcp}\ \{\Gamma \mid \neg\, (\Gamma \cup \mathrm{Axs}_1 \vdash_H \bot)\}.$$

  This requires a significant amount of manual effort for proving derived rules, e.g.

  $$\Gamma \cup \mathrm{Axs}_1 \vdash_H F \rightarrow G \longrightarrow \Gamma \cup \mathrm{Axs}_1 \vdash_H \neg\, F \vee G$$

  in our Hilbert system.

10 Conclusion

We have presented the formalization of a broad spectrum of calculi and results for classical propositional logic. Although all of the constructions and proofs we formalized "worked" in principle, the distances between the informal starting points and the formal text varied considerably. On one end of the spectrum were beautiful abstract results like the Model Existence Theorem whose proofs could be formalized very easily. On the other end of the spectrum was the translation from sequent calculus proofs into resolution refutations because it required to relate CNFs represented as formulas to CNFs represented as sets of clauses. Proofs about resolution graphs can also become more complicated if they take the form of global modifications of the graph: such one-step proofs required more subtle inductive arguments. Somewhere in the middle of the spectrum are proofs about sequent calculus, e.g. admissibility of cut or syntactic Craig interpolation. These careful syntactic arguments either lead to long manual proofs or require special purpose automation to deal with multisets (or structural rules, depending on the formalization).

This work is a first step towards a basis for the growing collection of formalized logical calculi. There is a plethora of important but non-trivial extensions, e.g. first-order, intuitionistic, or modal logics, that we hope to see formalized. Thiemann *et al.*'s formalization IsaFoR/CeTA [46] of large parts of rewriting theory (starting with [47]) shows that the dream of a unified formalization of logic is achievable and ideally the two efforts will be linked one day.

References

1 Stefan Berghofer. First-Order Logic According to Fitting. *Archive of Formal Proofs*, 2007. , Formal proof development. URL: http://isa-afp.org/entries/FOL-Fitting.shtml.

2 Jasmin C. Blanchette, Sascha Böhme, and Lawrence C. Paulson. Extending Sledgehammer with SMT Solvers. In N. Bjørner and V. Sofronie-Stokkermans, editors, *Automated Deduction (CADE 2011)*, volume 6803 of *LNCS*, pages 116–130. Springer, 2011.

3 Jasmin C. Blanchette et al. IsaFoL: Isabelle Formalization of Logic. URL: https://bitbucket.org/isafol/isafol.

4 Jasmin C. Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A Verified SAT Solver Framework with Learn, Forget, Restart, and Incrementality. *Accepted for publication in J. of Automated Reasoning*, 2018.

5 Jasmin C. Blanchette, Mathias Fleury, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. In N. Olivetti and A. Tiwari, editors, *Automated Reasoning (IJCAR 2016)*, volume 9706 of *LNCS*, pages 25–44. Springer, 2016.

6 Jasmin C. Blanchette, Andrei Popescu, and Dmitriy Traytel. Unified Classical Logic Completeness. In S. Demri, D. Kapur, and C. Weidenbach, editors, *Automated Reasoning (IJCAR 2014)*, volume 8562 of *LNCS*, pages 46–60. Springer, 2014.

7 Ana Bove, Alexander Krauss, and Matthieu Sozeau. Partiality and recursion in interactive theorem provers - an overview. *Mathematical Structures in Computer Science*, 26(1):38–88, 2016. doi:10.1017/S0960129514000115.

8 Patrick Braselmann and Peter Koepke. Gödel's Completeness Theorem. *Formalized Mathematics*, 13(**1**):49–53, 2005. URL: http://fm.mizar.org/2005-13/pdf13-1/goedelcp.pdf.

9 Peter Chapman, James McKinna, and Christian Urban. Mechanising a Proof of Craig's Interpolation Theorem for Intuitionistic Logic in Nominal Isabelle. In *Proc. AISC/Calculemus/MKM Conference*, pages 38–52. Springer, 2008.

10 Kaustuv Chaudhuri, Leonardo Lima, and Giselle Reis. Formalized Meta-Theory of Sequent Calculi for Substructural Logics. In *Workshop on Logical and Semantic Frameworks, with Applications (LSFA-11)*, 2016.

11 Christian Doczkal and Gert Smolka. Completeness and Decidability Results for CTL in Constructive Type Theory. *Automated Reasoning*, 56(3):343–365, 2016.

12 Herbert Enderton. *A Mathematical Introduction to Logic.* Academic Press, 1972.

13 Melvin Fitting. *First-Order Logic and Automated Theorem Proving.* Springer, 1990.

14 Andrew Gacek. *A Framework for Specifying, Prototyping, and Reasoning about Computational Systems.* PhD thesis, University of Minnesota, 2009.

15 Jean H Gallier. *Logic for computer science: foundations of automatic theorem proving.* Harper & Row, 1986.

16 John Harrison. Formalizing Basic First Order Model Theory. In J. Grundy and M. Newey, editors, *Theorem Proving in Higher Order Logics (TPHOLs '98)*, volume 1497 of *LNCS*, pages 153–170. Springer, 1998.

17 John Harrison. Towards self-verification of HOL Light. In U. Furbach and N. Shankar, editors, *Automated Reasoning (IJCAR 2006)*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

18 John Harrison. *Handbook of Practical Logic and Automated Reasoning*. Cambridge University Press, 2009.

19 David Hilbert. *Die Grundlagen der Mathematik*. Vieweg+Teubner Verlag, Wiesbaden, 1928. `doi:10.1007/978-3-663-16102-8_1`.

20 Michael Huth and Mark Ryan. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press, 2004.

21 Danko Illik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, 2010.

22 Alexander Krauss. Recursive Definitions of Monadic Functions. In Ekaterina Komendantskaya, Ana Bove, and Milad Niqui, editors, *Partiality and Recursion in Interactive Theorem Provers, PAR@ITP 2010*, volume 5 of *EPiC Series*, pages 1–13. EasyChair, 2012. URL: http://www.easychair.org/publications/?page=2065850452.

23 Ramana Kumar, Rob Arthan, Magnus Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic - Semantics, Soundness, and a Verified Implementation. *Autom. Reasoning*, 56(3):221–259, 2016.

24 Stephane Lescuyer. *Formalizing and Implementing a Reflexive Tactic for Automated Deduction in Coq*. PhD thesis, Université Paris Sud - Paris XI, 2011.

25 Filip Marić. Formalization and implementation of modern SAT solvers. *Automated Reasoning*, 43(1):81–119, 2009.

26 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.

27 Tobias Nipkow. Linear Quantifier Elimination. *Automated Reasoning*, 45:189–212, 2010.

28 Tobias Nipkow and Gerwin Klein. *Concrete Semantics with Isabelle/HOL*. Springer, 2014. URL: http://concrete-semantics.org.

29 Tobias Nipkow, Lawrence Paulson, and Markus Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

30 Duckki Oe, Aaron Stump, Corey Oliver, and Kevin Clancy. versat: A verified modern SAT solver. In V. Kuncak and A. Rybalchenko, editors, *Verification, Model Checking, and Abstract Interpretation (VMCAI 2012)*, volume 7148 of *LNCS*, pages 24–38. Springer, 2012.

31 Lawrence C. Paulson. A Mechanised Proof of Gödel's Incompleteness Theorems Using Nominal Isabelle. *Autom. Reasoning*, 55(1):1–37, 2015.

32 Frank Pfenning. Elf: A Language for Logic Definition and Verified Metaprogramming. In *Logic in Computer Science (LICS 1989)*, pages 313–322. IEEE Computer Society Press, 1989.

33 Frank Pfenning. Structural Cut Elimination: I. Intuitionistic and Classical Logic. *Inf. Comput.*, 157(1-2):84–141, 2000.

34 Frank Pfenning and Carsten Schürmann. System description: Twelf — A Meta-Logical Framework for Deductive Systems. In H. Ganzinger, editor, *Automated Deduction — CADE-16*, volume 1632 of *LNCS*, pages 202–206. Springer, 1999.

35 Brigitte Pientka and Andrew Cave. Inductive Beluga: Programming Proofs. In A. Felty and A. Middeldorp, editors, *Automated Deduction (CADE-25)*, volume 9195 of *LNCS*, pages 272–281. Springer, 2015.

36 Tom Ridge and James Margetson. A Mechanically Verified, Sound and Complete Theorem Prover for First Order Logic. In J. Hurd and T. Melham, editors, *Theorem Proving in Higher Order Logic (TPHOLs 2005)*, volume 3603 of *LNCS*, pages 294–309. Springer, 2005.

37 Anders Schlichtkrull. Formalization of the Resolution Calculus for First-Order Logic. In J. Blanchette and S. Merz, editors, *Interactive Theorem Proving (ITP 2016)*, volume 9807 of *LNCS*, pages 341–357. Springer, 2016.

38 Anders Schlichtkrull. Formalization of the Resolution Calculus for First-Order Logic. *J. of Automated Reasoning*, January 2018. `doi:10.1007/s10817-017-9447-z`.

39 Julian J. Schlöder and Peter Koepke. The Gödel completeness theorem for uncountable languages. *Formalized Mathematics*, 20(**3**):199–203, 2012. `doi:10.2478/v10037-012-0023-z`.

40 Uwe Schöning. *Logic for Computer Scientists.* Birkhäuser, 1989.

41 Kurt Schütte. Schlußweisen-Kalküle der Prädikatenlogik. *Mathematische Annalen*, 122(1):47–65, 1950. `doi:10.1007/BF01342950`.

42 Natarajan Shankar. *Metamathematics, Machines, and Gödel's Proof.* Cambridge University Press, 1994.

43 Natarajan Shankar and Marc Vaucher. The Mechanical Verification of a DPLL-Based Satisfiability Solver. *Electr. Notes Theor. Comput. Sci.*, 269:3–17, 2011.

44 Raymond M Smullyan. A unifying principal in quantification theory. *Proceedings of the National Academy of Sciences*, 49(6):828–832, 1963.

45 Raymond M. Smullyan. *First-Order Logic.* Springer, 1968.

46 Christian Sternagel, René Thiemann, et al. IsaFoR/CeTA: An Isabelle/HOL formalization of rewriting for certified termination analysis. URL: http://cl-informatik.uibk.ac.at/software/ceta/.

47 René Thiemann and Christian Sternagel. Certification of Termination Proofs Using CeTA. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logic (TPHOLs 2009)*, volume 5674 of *LNCS*, pages 452–468. Springer, 2009.

48 A. S. Troelstra and H. Schwichtenberg. *Basic Proof Theory.* Cambridge University Press, 2nd edition, 2000.

49 Christian Urban and Bozhi Zhu. Revisiting Cut-Elimination: One Difficult Proof Is Really a Proof. In A. Voronkov, editor, *Rewriting Techniques and Applications (RTA 2008)*, volume 5117 of *LNCS*, pages 409–424. Springer, 2008.

50 Kumar Neeraj Verma, Jean Goubault-Larrecq, Sanjiva Prasad, and S. Arun-Kumar. Reflecting BDDs in Coq. In He Jifeng and Masahiko Sato, editors, *ASIAN 2000: 6th Asian Computing Science Conference*, volume 1961 of *LNCS*, pages 162–181. Springer, 2000.

51 Bruno Woltzenlogel Paleo, editor. *Towards an Encyclopaedia of Proof Systems.* College Publications, London, UK, 1 edition, January 2017. URL: https://github.com/ProofSystem/Encyclopedia/blob/master/main.pdf.

Decomposing the Univalence Axiom

Ian Orton[1]
University of Cambridge Dept. Computer Science & Technology, Cambridge, UK
rio22@cam.ac.uk
https://orcid.org/0000-0002-9924-0623

Andrew M. Pitts
University of Cambridge Dept. Computer Science & Technology, Cambridge, UK
andrew.pitts@cl.cam.ac.uk
https://orcid.org/0000-0001-7775-3471

Abstract

This paper investigates Voevodsky's univalence axiom in intensional Martin-Löf type theory. In particular, it looks at how univalence can be derived from simpler axioms. We first present some existing work, collected together from various published and unpublished sources; we then present a new decomposition of the univalence axiom into simpler axioms. We argue that these axioms are easier to verify in certain potential models of univalent type theory, particularly those models based on cubical sets. Finally we show how this decomposition is relevant to an open problem in type theory.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases dependent type theory, homotopy type theory, univalent type theory, univalence, cubical type theory, cubical sets

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.6

Related Version `https://arxiv.org/abs/1712.04890`

Supplement Material `https://doi.org/10.17863/CAM.25036` (Agda 2.5.4 source code)

Acknowledgements We would like to thank the anonymous referees for their insightful comments which materially improved the quality of the paper.

1 Introduction

Extensionality is a principle whereby two mathematical objects are deemed to be equal if they have the same observable properties. Often, formal systems for mathematics will include axioms designed to capture this principle. In the context of set theory we have the axiom of extensionality, which tells us that two sets are equal if they contain the same elements. In the context of (univalent) type theory we have Voevodsky's univalence axiom [15, Section 2.10], which tells us, roughly speaking, that two types are equal if they are isomorphic.

In axiomatic set theory the axiom of extensionality is easily formalised as a simple implication $\forall A \forall B ((\forall X (X \in A \iff X \in B)) \implies A = B)$. The converse implication follows from the properties of equality, and combining these two implications we deduce that equality of two sets is logically equivalent to them having the same elements. At this point we are done, we now have an extensionality principle for sets and nothing further needs to be assumed.

[1] Supported by a UK EPSRC PhD studentship, funded by grants EP/L504920/1, EP/M506485/1.

The situation is more complicated in a proof relevant setting such as intensional type theory. As with sets we can formalise the statement of interest in the language of type theory as $(A\,B : U) \to A \simeq B \to A = B$, where $A \simeq B$ is the type of *equivalences* between A and B. We then postulate the existence of a term witnessing this statement. As before, the converse implication follows from the properties of the identity type, and hence equality of two types is logically equivalent to them being isomorphic. However, the proof relevant nature of type theory means that what we have described so far will be insufficient. We may want to know how equalities derived using the postulated term compute when passed to the eliminator for identity types. For example, if we convert them back into equivalences do we always get the same equivalence that we started with?

In univalent type theory (UTT), also known as homotopy type theory (HoTT), these problems are resolved by taking a different approach to the statement of the univalence axiom. As mentioned before the converse implication, $(A\,B : U) \to A = B \to A \simeq B$, follows from the properties of the identity type. The approach taken in UTT is to state that for any types A and B this map is itself an equivalence between the types $A = B$ and $A \simeq B$. From this fact we can deduce the existence of a map in the other direction (the original implication of interest), as well as some information about how that map computes.

Merely stating that a certain canonical map is an equivalence is a very concise way to express the univalence axiom. From a mathematical point of view it is appealingly simple and yet powerful. In particular, this statement has the nice property that it is a *mere proposition* [15, Definition 3.3.1] and so there is no ambiguity about the term witnessing the axiom.

However, there are some disadvantages to this way of stating the univalence axiom. For example, verifying the univalence axiom in a model of type theory can be a difficult task. Fully expanded, this seemingly simple statement becomes very large, with many complex subterms. Verifying univalence directly, by computing the interpretation of the statement in the model and explicitly constructing the interpretation of its proof term, may be unfeasible. Instead, one would need to build up several intermediate results about contractibility, equivalences, and possibly new constructions such as *Glueing* [8, Section 6], through a mixture of internal, syntactic and semantic arguments.

The contribution in this paper is a reduction of the usual statement of univalence to a collection of simpler axioms which are more easily verified in certain models of dependent type theory, particularly those based on cubical sets [8, 5, 2, 3, 14, 6]. Importantly, we do not propose these axioms as an alternative statement for the univalence axiom when doing mathematics in univalent type theory. These axioms are designed with the previous goal in mind and are not intended to be mathematically elegant or user-friendly.

In the rest of this paper we begin with some preliminary definitions and notational conventions (Section 2). We then briefly discuss the univalence axiom (Section 3). These sections cover existing work. We then introduce our alternative set of axioms (Section 4), and examine their application to models of type theory (Section 5). Finally, we propose another application of these axioms to an open problem in UTT (Section 6).

Agda formalisation

This work presented in this paper is supported by two separate developments in the Agda proof assistant [1]. The first covers the material in sections 2-4, where Agda is useful for precisely tracking universe levels in many of the theorems. The second covers the material in Section 5, and builds on the development accompanying [13]. In this development we use Agda to verify our constructions in the internal type theory of the cubical sets topos. The source for both can be found at `https://doi.org/10.17863/CAM.25036`.

2 Preliminaries

In most of this paper we work in intensional Martin-Löf type theory with dependent sums and products, intentional identity types, and a cumulative hierarchy of universes $U_0 : U_1 : U_2 : \ldots$.

We use the symbol $=$ for the identity type, $\equiv$ for definitional equality and $\triangleq$ when giving definitions. Given $p : x = y$ and $q : y = z$, we write $p \cdot q : x = z$ for the composition of identity proofs, and $p^{-1} : y = x$ for the inverse proof.

We also assume the principle of function extensionality, which states that two functions $f, g : \prod_{x:A} B(x)$ are equal whenever they are pointwise equal: $f \approx g \triangleq \prod_{x:A} f(x) = g(x)$. That is, that there exists a term:

$$\mathit{funext}_{i,j} : \prod_{A:U_i} \prod_{B:A \to U_j} \prod_{f,g:\Pi_{x:A} B(x)} f \approx g \to f = g$$

for all universe levels i, j.

Note that in the Agda development mentioned previously we do not assume function extensionality in general, but rather we make it an explicit assumption to each theorem. This means that we can see exactly where function extensionality is used, and at which universe levels it needs to hold.

We now recall some standard definitions and results in UTT/HoTT.

▶ **Definition 2.1** (Contractibility). A type A is said to be *contractible* if the type

$$\mathit{isContr}(A) \triangleq \sum_{a_0:A} \prod_{a:A} (a_0 = a)$$

is inhabited. Contractibility expresses the fact that a type has a unique inhabitant.

▶ **Definition 2.2** (Singletons). Given a type A and element $a : A$, we can define

$$\mathit{sing}(a) \triangleq \sum_{x:A} (a = x)$$

to be the type of elements of A which are equal to a. It is easily shown by path induction that the type $\mathit{sing}(a)$ is always contractible.

▶ **Definition 2.3** (Equivalences). An *equivalence* from $A \simeq B$ is a pair (f, e) where $f : A \to B$ and e is a proof that for every $b : B$ the fiber of f at b is contractible. To be precise:

$$A \simeq B \triangleq \sum_{f:A \to B} \mathit{isEquiv}(f)$$

where

$$\mathit{fib}_f(b) \triangleq \sum_{a:A} (f\ a = b) \qquad \text{and} \qquad \mathit{isEquiv}(f) \triangleq \prod_{b:B} \mathit{isContr}(\mathit{fib}_f(b))$$

for $A : U_i$, $B : U_j$ for any i, j.

A simple example of an equivalence is the identity function $\mathit{id}_A : A \to A$ for any type A. To demonstrate that id_A is an equivalence we must show that $\prod_{a:A} \mathit{isContr}(\sum_{x:A}(a = x))$, but this is equivalent to the statement that $\mathit{sing}(a)$ is contractible for all $a : A$.

3 Voevodsky's Univalence Axiom

In this section we introduce Voevodsky's univalence axiom. We then present an existing result which decomposes the univalence axiom into a "naive" form and a computation rule. In Section 4, we will then decompose these two axioms further into five even simpler axioms.

▶ **Definition 3.1** (Coerce and idtoeqv)**.** For all i, and types $A, B : U_i$, there is a canonical map $\mathit{idtoeqv} : (A = B) \to (A \simeq B)$ which is defined by path induction on the proof $A = B$:

$$\mathit{idtoeqv}(\mathtt{refl}) \triangleq \mathit{id}_A$$

where $\mathit{id}_A : A \simeq A$ is the identity map regarded as an equivalence. We can also define a map $\mathtt{coerce} : (A = B) \to A \to B$ either by path induction, or as:

$$\mathtt{coerce}(p, a) \triangleq \mathtt{fst}(\mathit{idtoeqv}(p))(a)$$

where $\mathtt{fst}$ is the first projection.

▶ **Definition 3.2** (Voevodsky's univalence axiom)**.** The univalence axiom for a universe U_i asserts that for all $A, B : U_i$ the map $\mathit{idtoeqv} : (A = B) \to (A \simeq B)$ is an equivalence.

In light of the following definition we will often refer to the univalence axiom as the *proper* univalence axiom.

▶ **Definition 3.3** (The naive univalence axiom)**.** The naive univalence axiom for a universe U_i gives, for all $A, B : U_i$, a map from equivalences to equalities. In other words, it asserts the existence of an inhabitant of the type:

$$UA_i \triangleq \prod_{A,B : U_i} A \simeq B \to A = B$$

When using a term $ua : UA_i$ we will often omit the first two arguments (A and B). Proofs of naive univalence may also come with an associated computation rule. That is, an inhabitant of the type $UA\beta_i(ua)$, where:

$$UA\beta_i(ua) \triangleq \prod_{A,B : U_i} \prod_{f : A \to B} \prod_{e : \mathit{isEquiv}(f)} \mathtt{coerce}\ (ua(f, e)) = f$$

Next, we give a result which is known in the UTT/HoTT community and has been discussed on the HoTT mailing list. However, the authors are not aware of any existing presentation of a proof in the literature. This result decomposes the proper univalence axiom into the naive version and a computation rule. First we give a lemma which generalises the core construction of this result.

▶ **Lemma 3.4.** *Given $X : U_i$, $Y : X \to X \to U_j$ and a map $f : \prod_{x,x' : X} x = x' \to Y(x, x')$ then $f\, x\, x'$ is an equivalence for all $x, x' : X$ iff there exists a map*

$$g : \prod_{x,x' : X} Y(x, x') \to x = x'$$

such that for all $x, x' : X$ and $y : Y(x, x')$ we have $f(g(y)) = y$ (we leave the first two arguments to f and g implicit).

Table 1 ($A, B : U_i$, $C : A \to B \to U_i$, $a : A$, $b : B$ and $c : C\,a\,b$, for some universe U_i).

	Axiom		Premise(s)		Equality		
(1)	*unit*	:			A	$=$	$\sum_{a:A} 1$
(2)	*flip*	:			$\sum_{a:A}\sum_{b:B} C\ a\ b$	$=$	$\sum_{b:B}\sum_{a:A} C\ a\ b$
(3)	*contract*	:	*isContr* A	$\to$	A	$=$	1
(4)	*unit*β	:			*coerce unit* a	$=$	$(a, *)$
(5)	*flip*β	:			*coerce flip* (a, b, c)	$=$	(b, a, c)

Proof. For the backwards direction, assume that we are given g as above. To show that f is an equivalence it suffices to show that f is a bi-invertible map [15, Section 4.3]. To do this we must exhibit both a right and left inverse.

For the left inverse we take $g'(y) \triangleq g(y) \cdot g(f(\texttt{refl}))^{-1}$. To see that this is indeed a left inverse to f consider an arbitrary $p : x = x'$, we aim to show that $g'(f(p)) = p$. By path induction we may assume that $x \equiv x'$ and $p \equiv \texttt{refl}$ and therefore we are required to show $g'(f(\texttt{refl})) = \texttt{refl}$. However, since $g'(f(\texttt{refl})) \equiv g(f(\texttt{refl})) \cdot g(f(\texttt{refl}))^{-1}$ this goal simplifies to $g(f(\texttt{refl})) \cdot g(f(\texttt{refl}))^{-1} = \texttt{refl}$ which follows immediately from the groupoid laws for identity types.

For the right inverse we take g unchanged and observe that we know $f(g(y)) = y$ for all $y : Y(x, x')$ by assumption. Therefore the map f is an equivalence.

For the forwards direction, given a proof $e : \mathit{isEquiv}(f)$ and $y : Y(x, x')$ we have $\texttt{fst}(e(y)) : \sum_{p:x=x'} f(p) = y$. We can then define $g(y)$ to be the first component of this and the second component tells us that $f(g(y)) = y$ as required. ◀

▶ **Theorem 3.5.** *Naive univalence, along with a computation rule, is logically equivalent to the proper univalence axiom. That is, there are terms*

$$ua : UA_i, \qquad ua\beta : UA\beta_i(ua)$$

iff for all types $A, B : U_i$, the map $\mathit{idtoeqv} : (A = B) \to (A \simeq B)$ is an equivalence.

Proof. By $ua\beta$ we know that $\texttt{fst}(\mathit{idtoeqv}(ua(f, e))) = f$ for all $(f, e) : A \simeq B$. Now, since $\mathit{isEquiv}(f)$ is a mere proposition for each f, we can deduce that $\mathit{idtoeqv}(ua(f, e)) = (f, e)$ by [15, Lemma 3.5.1]. Therefore we simply take $X \equiv U_i$, $Y(A, B) \equiv A \simeq B$, $f \equiv \mathit{idtoeqv}$ and $g \equiv ua$ in Lemma 3.4 to deduce the desired result. ◀

4 A new set of axioms

In this section we further decompose the univalence axiom into even simpler axioms. We show that it is equivalent to axioms (1) to (5) given in Table 1. Note that these axioms apply to a specific universe U_i.

We begin by decomposing naive univalence, UA_i, into axioms (1)-(3). These axioms also follow from UA_i. Recall that we are taking function extensionality as an ambient assumption.

▶ **Theorem 4.1.** *Axioms (1)-(3) for a universe U_i are together logically equivalent to UA_i.*

Proof. We begin by showing the forwards direction. Assume that we are given axioms (1) to (3). We now aim to define a term $ua : UA_i$. Given arbitrary types $A, B : U_i$ and an equivalence $(f, e) : A \simeq B$ we define $ua(f, e) : A = B$ as follows:

$$\begin{aligned}
A &= \sum_{a:A} 1 && \text{by (1)} \\
&= \sum_{a:A} \sum_{b:B} f\ a = b && \text{by funext and (3) on } sing(fa) \\
&= \sum_{b:B} \sum_{a:A} f\ a = b && \text{by (2)} \\
&= \sum_{b:B} 1 && \text{by funext and (3) on } fib_f(b) \text{ (contractible by } e) \\
&= B && \text{by (1)}
\end{aligned}$$

where the proof that $A = B$ is given by the concatenation of each step of the above calculation.

The backwards direction follows from the fact that the obvious maps $A \to \sum_{a:A} 1$ and $(\sum_{a:A} \sum_{b:B} C\ a\ b) \to (\sum_{b:B} \sum_{a:A} C\ a\ b)$ are both easily shown to be bi-invertible and hence equivalences, and from the fact that any contractible type is equivalent to 1 [15, Lemma 3.11.3.]. Therefore given $ua : UA_i$ we simply apply it to these equivalences to get the required equalities (1)-(3). ◀

Next, we decompose the computation rule for naive univalence $UA\beta_i$ into axioms (4) and (5). Since $UA\beta_i$ depends on UA_i and axioms (4) and (5) depend on axioms (1) and (2) respectively, we in fact show the logical equivalence between the pair UA_i and $UA\beta_i$, and axioms (1)-(5).

▶ **Lemma 4.2.** *The function* `coerce` *is compositional. That is, given types $A, B, C : U_i$, and equalities $p : A = B$ and $q : B = C$ we have* $\texttt{coerce}(p \cdot q) = \texttt{coerce}(q) \circ \texttt{coerce}(p)$.

Proof. Straightforward by path induction on either of p or q, or on both. ◀

▶ **Theorem 4.3.** *Axioms (1)-(5) for a universe U_i are together logically equivalent to $\sum_{ua:UA_i} UA\beta_i(ua)$.*

Proof. For the forwards direction we know from Theorem 4.1 that axioms (1) to (3) allow us to construct a term $ua : UA_i$. If, in addition, we assume axioms (4) and (5) then we can show that for all $(f, e) : A \simeq B$ we have $\texttt{coerce}(ua(f, e)) = f$ as follows.

Since ua was constructed as the concatenation of five equalities then, in light of Lemma 4.2, we have that coercing along $ua(f, e)$ is equal to the result of coercing along each stage of the composite equality $ua(f, e)$. Therefore starting with an arbitrary $a : A$, we can track what happens at each stage of this process like so:

$$a \quad \mapsto \quad (a,\ *) \quad \mapsto \quad (a,\ f\ a,\ \mathit{refl}) \quad \mapsto \quad (f\ a,\ a,\ \mathit{refl}) \quad \mapsto \quad (f\ a,\ *) \quad \mapsto \quad f\ a$$

Therefore we see that for all $a : A$ we have $\texttt{coerce}(ua(f, e))(a) = f(a)$ and hence by function extensionality we have $\texttt{coerce}(ua(f, e)) = f$ as required.

For the reverse direction we assume that we are given $ua : UA_i$ and $ua\beta : UA\beta_i(ua)$. We can now apply Theorem 4.1 to construct terms *unit*, *flip* and *contract* satisfying axioms (1) to (3) from ua.

Since *unit* and *flip* were constructed by applying ua to the obvious equivalences, then by $ua\beta$ we know that applying `coerce` to these equalities will return the equivalences that we started with. From this we can easily construct terms $unit\beta$ and $flip\beta$ satisfying axioms (4) and (5) respectively. ◀

▶ **Corollary 4.4.** *Axioms (1)-(5) for a universe U_i are together logically equivalent to the proper univalence axiom for U_i.*

Proof. By combining Theorems 3.5 and 4.3. ◀

5 Applications in models of type theory

In this section we discuss one reason why the result given in Corollary 4.4 is useful when trying to construct models of univalent type theory. Specifically, we believe that this decomposition is particularly useful for showing that a model of type theory with an interval object (e.g. cubical type theory [8]) supports the univalence axiom. We first explain why we believe this to be the case in general terms, and then give a precise account of what happens in the specific case of the cubical sets model presented in [8]. The arguments given here should translate to many similar models of type theory [5, 2, 3, 14, 6].

Note that we are assuming function extensionality. Every model of univalence must satisfy function extensionality [15, Section 4.9], but it is often much easier to verify function extensionality than the proper univalence axiom in a model of type theory. In particular, function extensionality will hold in any type theory which includes an appropriate interval type, cf. [15, Lemma 6.3.2].

Experience shows that axioms (1), (2), (4) and (5) are simple to verify in many potential models of univalent type theory. To understand why, it is useful to consider the interpretation of $A \simeq B$ in such a model. Propositional equality in the type theory is usually not interpreted as equality in the model's metatheory, but rather as a construction on types e.g. path spaces in models of HoTT. Therefore, writing $[\![X]\!]$ for the interpretation of a type X, an equivalence in the type theory will give rise to morphisms $f : [\![A]\!] \to [\![B]\!]$ and $g : [\![B]\!] \to [\![A]\!]$ which are not exact inverses, but rather are inverses modulo the interpretation of propositional equality, e.g. the existence of paths connecting x and $g(f(x))$, and y and $f(g(y))$ for all $x \in [\![X]\!], y \in [\![Y]\!]$. However, in many models the interpretations of A and $\sum_{a:A} 1$, and of $\sum_{a:A} \sum_{b:B} C\ a\ b$ and $\sum_{b:B} \sum_{a:A} C\ a\ b$ will be isomorphic, i.e. there will be morphisms going back and forth which are inverses up to equality in the model's metatheory. This will be true in any presheaf model of type theory of the kind described in Section 5.1.1, and should be true more generally in any model which validates eta-rules for 1 and Σ.

This means that we can satisfy (1) and (2) by proving that this stronger notion of isomorphism gives rise to a propositional equality between types. Verifying axioms (4) and (5) should then reduce to a fairly straightforward calculation involving two instance of this construction.

This leaves axiom (3), which captures the homotopical condition that every contractible space can be continuously deformed into a point. The hope is that verifying the previous axioms should be fairly straightforward, leaving this as the only non-trivial condition to check.

We now examine what happens in the specific case of cubical sets [8].

5.1 Example: the CCHM model of univalent type theory

In this section we will examine what happens in the case of the Cohen, Coquand, Huber, Mörtberg (CCHM) model of type theory based on cubical sets [8]. To be clear, this model is shown to validate the univalence axiom in the previously cited paper. However, here we give an alternative, hopefully simpler, proof of univalence using the decomposition given in Section 4. We start from the knowledge that cubical sets model a type theory with *Path* types given by maps out of an interval object $\mathtt{I}$, and where types come equipped with a *composition* operation which is closed under all type formers $(\Sigma, \Pi, Path)$. From this we then show how to validate our axioms, and therefore the proper univalence axiom. This therefore yields an alternative proof of univalence which does not use the glueing construction from [8]. Note however that this construction is still required to show that the universe is fibrant, and hence we cannot completely eliminate the use of glueing from the CCHM model.

For most of this section we will work in the internal language of the cubical sets topos, using a technique developed by the authors in a previous paper [13]. We begin with a brief summary of the cubical sets model and then describe the internal language approach to working with such models. For those unfamiliar with this material we refer the reader to [8] and [13] respectively for further details.

5.1.1 The cubical sets model

Cohen *et al* [8] present a model of type theory using the category $\hat{\mathcal{C}}$ of presheaves on the small category $\mathcal{C}$ whose objects are given by finite sets of symbols, written I, J, K, with $\mathcal{C}(I, J)$ being the set of maps $J \to dm(I)$, where $dm(I)$ is the free De Morgan algebra [4] on the set I.

First we recall the standard way of constructing a presheaf model of type theory [10], note that this is not the final model construction. Take $\hat{\mathcal{C}}$ to be the category of contexts with the types over a context $\Gamma \in \hat{\mathcal{C}}$, written $\mathtt{Ty}(\Gamma)$, given by presheaves on Γ's category of elements. Terms of type $A \in \mathtt{Ty}(\Gamma)$, written $Ter(\Gamma \vdash A)$ are simply global sections of A. Explicitly, this means that a type $A \in \mathtt{Ty}(\Gamma)$ is given by a family of sets $A(I, \rho)$ for every $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ such that for every $a \in A(I, \rho)$ and $f : J \to I$ we have $A(f)(a) \in A(J, \Gamma(f)(\rho))$ with $A(id_I)(a) = a$ and $A(g \circ f)(a) = A(f)(A(g)(a))$. A term $a \in Ter(\Gamma \vdash A)$ is given by a family $a(I, \rho) \in A(I, \rho)$ for every $I \in \mathcal{C}$ and $\rho \in \Gamma(I)$ such that for all $f : J \to I$ we have $A(f)(a(I, \rho)) = a(J, \Gamma(f)(\rho))$. Following the convention in [8] we will often omit the first argument I and will write functorial actions $\Gamma(f)(\rho)$ simply as ρf.

These constructions all model substitution, context extension, projection, etc, in the correct way, and can be shown to form a category with families (CwF) in the sense of Dybjer [9]. Such a model always supports both dependent sums and products. For example, given types $A \in \mathtt{Ty}(\Gamma)$ and $B \in \mathtt{Ty}(\Gamma.A)$ then the dependent sum $\Sigma AB \in \mathtt{Ty}(\Gamma)$ can be interpreted as

$$\Sigma AB(I, \rho) \ \triangleq\ \{(a, b) \mid a \in A(I, \rho),\ b \in B(I, \rho, a)\}$$

As stated in the previous section, any model of this kind will always have $\Sigma A(\Sigma BC)$ being strictly isomorphic to $\Sigma B(\Sigma AC)$, that is, with natural transformations in each direction which are inverses up to equality in the model's metatheory. Furthermore, assuming that the terminal type 1 is interpreted as the terminal presheaf, then the same will be true for the types A and $\Sigma A1$. This is potentially useful when verifying axioms (1), (2), (4) and (5) for the reasons given above.

To get a model of type theory which validates the univalence axiom we restrict our attention to types with an associated *composition structure* [8, Definition 13]. We call such types *fibrant* and write $\mathtt{FTy}(\Gamma)$ for the collection of fibrant types over a context $\Gamma \in \hat{\mathcal{C}}$, and $\mathtt{FTy}_0(\Gamma)$ for the collection of small types. Taking contexts, terms, type formers and substitution as before, we get a new CwF of fibrant types. We delay giving the exact definition of a composition structure until after we introduce the internal type theory of $\hat{\mathcal{C}}$ in the following section.

5.1.2 The internal type theory of $\hat{\mathcal{C}}$

In previous work [13] the authors axiomatised the properties of the cubical sets topos needed to develop a model of univalent type theory. They then showed how many of the constructions used in the model could be replicated using the internal type theory of an elementary topos [12]. Here we will often build on this approach, working mostly in the internal type theory. We now give a brief overview of this approach, and refer the reader to [13] for full details.

We use a concrete syntax inspired by Agda [1]. Dependent function types are written as $(x : A) \to B$ with lambda abstractions written as $\lambda(x : A) \to t$. We use $\{\}$ in place of $()$ to indicate the use of implicit arguments. Dependent product types are written as $(x : A) \times B$ with the pairing operation written as (s, t).

We assume the existence of an interval object $\mathtt{I}$ with endpoints $\mathtt{0}, \mathtt{1} : 1 \to \mathtt{I}$ subject to certain conditions, a class of propositions $\mathtt{Cof} \rightarrowtail \Omega$, closed under $\vee, \wedge$ and $\mathtt{I}$-indexed $\forall$, which we call the *cofibrant* propositions and two internal Russell-style universes $\mathcal{U} : \mathcal{U}_1$, closed under all the type formers of the internal language. Given $\varphi : \Omega$ we write $[\varphi] \triangleq \{\_ : 1 \mid \varphi\}$ for the type whose inhabitation corresponds to the provability of φ, and given a object $\Gamma : \mathcal{U}$ and a cofibrant property $\Phi : \Gamma \to \mathtt{Cof}$ we write $\Gamma|\Phi \triangleq (x : \Gamma) \times [\Phi\, x]$ for the *restriction of* Γ *by* Φ. Given $\varphi : \mathtt{Cof}$, $f : [\varphi] \to A$ and $a : A$ we write $(\varphi, f) \nearrow a$ for $(u : [\varphi]) \to f\, u = a$; thus elements of this type are proofs that the partial element f (with cofibrant domain of definition φ) extends to the totally defined element a. Finally, we write $a \sim a' \triangleq \{p : \mathtt{I} \to A \mid p\, \mathtt{0} = a \wedge p\, \mathtt{1} = a'\}$ for the type of *paths* from $a : A$ to $a' : A$.

As an example of the use of this language we now reproduce the internal definition of a (small) fibration [13, Definition 5.7]:

▶ **Definition 5.1** (CCHM fibrations). A *CCHM fibration* (A, α) over a type $\Gamma : \mathcal{U}$ is a family $A : \Gamma \to \mathcal{U}$ equipped with a fibration structure $\alpha : \mathtt{isFib}\, A$, where $\mathtt{isFib} : \{\Gamma : \mathcal{U}\}(A : \Gamma \to \mathcal{U}) \to \mathcal{U}$ is defined by

$$\mathtt{isFib}\, \{\Gamma\}\, A \triangleq (e : \{\mathtt{0},\mathtt{1}\})(p : \mathtt{I} \to \Gamma) \to \mathtt{Comp}\, e\, (A \circ p)$$

Here $\mathtt{Comp} : (e : \{\mathtt{0},\mathtt{1}\})(A : \mathtt{I} \to \mathcal{U}) \to \mathcal{U}$ is the type of *composition structures* for $\mathtt{I}$-indexed families:

$$\begin{aligned}\mathtt{Comp}\, e\, A \triangleq\ & (\varphi : \mathtt{Cof})(f : [\varphi] \to \Pi_{\mathtt{I}} A) \to \\ & \{a_0 : A\, e \mid (\varphi, f) @ e \nearrow a_0\} \to \{a_1 : A\, \overline{e} \mid (\varphi, f) @ \overline{e} \nearrow a_1\}\end{aligned}$$

where $(\varphi, f) @ e$ is an abbreviation for the term $\lambda u : [\varphi].\, f\, u\, e$ of type $[\varphi] \to A\, e$, and where $\overline{\mathtt{0}} = \mathtt{1}$ and $\overline{\mathtt{1}} = \mathtt{0}$.

We write $\mathtt{Fib}\, \Gamma \triangleq (A : \Gamma \to \mathcal{U}) \times \mathtt{isFib}\ A$ for the type of fibrations over $\Gamma : \mathcal{U}$ and recall that fibrations can be reindexed $(A, \alpha)[\gamma] = (A \circ \gamma, \alpha[\gamma]) : \mathtt{Fib}\ \Delta$ for $\gamma : \Delta \to \Gamma$.

5.1.3 Paths between fibrations

We work in the internal type theory of the cubical sets topos wherever possible, however this approach does have its limitations. In particular this internal approach is unable to describe type theoretic universes [13, Remark 7.5]. Therefore we will not be able to construct elements of the identity type on the universe (the target type of axioms (1)-(3)). Instead, we work with an (externally) equivalent notion of equality between types.

▶ **Definition 5.2** (Path equality between fibrations). Define the type of paths between CCHM fibrations $\_ \sim_{\mathcal{U}} \_ : \{\Gamma : \mathcal{U}\} \to \mathtt{Fib}\, \Gamma \to \mathtt{Fib}\, \Gamma \to \mathcal{U}_1$ by

$$(A, \alpha) \sim_{\mathcal{U}} (B, \beta) \triangleq \{(P, \rho) : \mathtt{Fib}(\Gamma \times \mathtt{I}) \mid (P, \rho)[\langle id, \mathtt{0}\rangle] = (A, \alpha) \wedge (P, \rho)[\langle id, \mathtt{1}\rangle] = (B, \beta)\}$$

To understand why this notion of path is equivalent to the usual notion recall that the universe construction in [8] is given by the usual Hofmann-Streicher universe construction for presheaf categories [11]. This means that there exists a type $\mathcal{U} \in \mathtt{Ty}(\Gamma)$ for all Γ given by $\mathcal{U}(I, \rho) \triangleq \mathtt{FTy}_0(\mathrm{y}I)$ where $\mathrm{y}I$ denotes the Yoneda embedding of I. Every (small) fibrant type $A \in \mathtt{FTy}_0(\Gamma)$ has a code $\ulcorner A \urcorner \in \mathit{Ter}(\Gamma \vdash \mathcal{U})$ and every $a \in \mathit{Ter}(\Gamma \vdash \mathcal{U})$ encodes a type $\mathit{El}\, a \in \mathtt{FTy}_0(\Gamma)$ such that $\mathit{El}\,(\ulcorner A \urcorner) = A$ and $\ulcorner \mathit{El}\, a \urcorner = a$ for all a and A.

Now consider the following: externally, a path $P : A \sim_{\mathcal{U}} B$ corresponds to a fibration $P \in \mathtt{FTy}_0(\Gamma.\mathbb{I})$ such that $P[\langle id, 0\rangle] = A$ and $P[\langle id, 1\rangle] = B$ for some $\Gamma \in \hat{\mathcal{C}}$ and $A, B \in \mathtt{FTy}_0(\Gamma)$. From this data we can construct $p \in \mathit{Ter}(\Gamma \vdash \mathit{Path}\, \mathcal{U}\, \ulcorner A \urcorner\, \ulcorner B \urcorner)$ like so:

$$p(\rho) \triangleq \langle i \rangle\, \ulcorner P \urcorner(\rho s_i, i)$$

for $I \in \mathcal{C}, \rho \in \Gamma(I)$, where $s_i : I, i \to I$ is the obvious inclusion of I in I, i. Note that this does define a path with the correct endpoints since substituting 0 for i we get:

$$(\ulcorner P \urcorner(\rho, 0))f = P(\rho f, 0f) = P(\rho f, 0) = A(\rho f) = (\ulcorner A \urcorner(\rho))f$$

for all $f : J \to I$. The case for $i = 1$ is similar.

Conversely, given $p \in \mathit{Ter}(\Gamma \vdash \mathit{Path}\, \mathcal{U}\, \ulcorner A \urcorner\, \ulcorner B \urcorner)$ we can define $P \in \mathtt{FTy}_0(\Gamma.\mathbb{I})$ with the required properties like so:

$$P(\rho, i) \triangleq (p\ \rho\ i) id_I$$

for $I \in \mathcal{C}, \rho \in \Gamma(I), i \in \mathbb{I}$. Again, note that this has the correct properties, e.g. at 0:

$$P[\langle id, 0\rangle]\ \rho = P(\rho, 0) = (p\ \rho\ 0)\ id_I = (\ulcorner A \urcorner\ \rho) id_I = (El \ulcorner A \urcorner)\rho = A\rho$$

for all $\rho \in \Gamma(I)$. It is easily checked that these two constructions are mutual inverses. Therefore the data described by $\_ \sim_{\mathcal{U}} \_$ corresponds exactly to the data required to describe a path in the universe.

5.1.4 The realignment lemma

Next, we introduce a technical lemma that will be needed in the following sections. For readers familiar with the cubical sets model, it is interesting to note that this is the only place where we use the fact that cofibrant propositions are closed under $\mathtt{I}$-indexed $\forall$ [8, Section 4.1].

► **Lemma 5.3** (Realignment lemma). *Given $\Gamma : \mathcal{U}$ and $\Phi : \Gamma \to \mathtt{Cof}$, let $\iota : \Gamma|\Phi \rightarrowtail \Gamma$ be the first projection. For any $A : \Gamma \to \mathcal{U}$, $\beta : \mathtt{isFib}(A \circ \iota)$ and $\alpha : \mathtt{isFib}\, A$, there exists a composition structure $\mathtt{realign}(\Phi, \beta, \alpha) : \mathtt{isFib}\, A$ such that $\beta = \mathtt{realign}(\Phi, \beta, \alpha)[\iota]$.*

Proof. By [13, Theorem 6.13], in which we define $\mathtt{realign}(\Phi, \beta, \alpha)$ by

$$\mathtt{realign}(\Phi, \beta, \alpha)\, e\, p\, \psi\, f\, a \triangleq \alpha\, e\, p\, (\psi \vee (\forall (i : \mathtt{I}).\, \Phi\, (p\, i)))\, (f \cup f')\, a \tag{1}$$

where $f' : [\forall (i : \mathtt{I}).\, \Phi\, (p\, i)] \to \Pi_{\mathtt{I}}(A \circ p)$ is given by $f'\, u \triangleq \mathtt{fill}\ e\, \beta\, (\lambda i \to (p, u\, i))\, \psi\, f\, a$. ◄

In words, given a fibrant type A and an alternative composition structure defined only on some restriction of A, then we can *realign* the original composition structure so that it agrees with the alternative on that restriction.

Note that this construction is stable under reindexing in the following sense: given $\gamma : \Delta \to \Gamma$, $\Phi : \Gamma \to \texttt{Cof}$, $A : \Gamma \to \mathcal{U}$, $\beta : \texttt{isFib}(A \circ \iota)$ and $\alpha : \texttt{isFib}\, A$ then,

$$\begin{aligned}
&\texttt{realign}(\Phi, \beta, \alpha)[\gamma]\, e\, p\, \psi\, f\, a \\
&\quad = \texttt{realign}(\Phi, \beta, \alpha)\, e\, (\gamma \circ p)\, \psi\, f\, a \\
&\quad = \alpha\, e\, (\gamma \circ p)\, (\psi \vee (\forall (i : \texttt{I}).\, \Phi\, ((\gamma \circ p)\, i)))\, (f \cup \texttt{fill}\ e\, \beta\, (\lambda i \to (\gamma \circ p, u\, i))\, \psi\, f\, a)\, a \\
&\quad = \alpha[\gamma]\, e\, p\, (\psi \vee (\forall (i : \texttt{I}).\, (\Phi \circ \gamma)(p\, i)))\, (f \cup \texttt{fill}\ e\, \beta[\langle \gamma, id \rangle]\, (\lambda i \to (p, u\, i))\, \psi\, f\, a)\, a \\
&\quad = \texttt{realign}(\Phi \circ \gamma, \beta[\langle \gamma, id \rangle], \alpha[\gamma])\, e\, p\, \psi\, f\, a
\end{aligned}$$

Therefore we have

$$\texttt{realign}(\Phi, \beta, \alpha)[\gamma] = \texttt{realign}(\Phi \circ \gamma, \beta[\langle \gamma, id \rangle], \alpha[\gamma])$$

5.1.5 Fibrations are closed under isomorphism

▶ **Definition 5.4** (Strict isomorphism). A *strict isomorphism* between two objects $A, B : \mathcal{U}$ is a pair (f, g) where $f : A \to B$ and $g : B \to A$ such that $g \circ f = id$ and $f \circ g = id$. We write $(f, g) : A \cong B$.

This notion lifts to both families and fibrations, and we overload the notation $\_ \cong \_$ like so: when $A, B : \Gamma \to \mathcal{U}$ then we take $A \cong B$ to mean $(x : \Gamma) \to A\, x \cong B\, x$ and when $A, B : \texttt{Fib}\, \Gamma$ then we take $A \cong B$ to mean $\texttt{fst}\, A \cong \texttt{fst}\, B$.

▶ **Lemma 5.5.** *Given a family $A : \Gamma \to \mathcal{U}$ and a fibration $(B, \beta) : \texttt{Fib}\, \Gamma$, such that $A \cong B$, then we can construct an α such that $(A, \alpha) : \texttt{Fib}\, \Gamma$.*

Proof. Assume that we are given A and (B, β) as above with an isomorphism $\langle f, g \rangle : A \cong B$. We can then define a composition structure for A as follows:

$$\alpha\ e\ p\ \varphi\ q\ a_0 \triangleq g\ (p\ \bar{e})\ (\beta\ e\ p\ \varphi\ (\lambda u\, i \to f\, (p\, i)\, (q\ u\ i))\ (f\ (p\, e)\, a_0))$$

This construction has the required property that, given $u : [\varphi]$:

$$\begin{aligned}
\alpha\ e\ p\ \varphi\ q\ a_0 &= g\ (p\ \bar{e})\ (\beta\ e\ p\ \varphi\ (\lambda u\, i \to f\, (p\, i)\, (q\ u\ i))\ (f\ (p\, e)\, a_0)) \\
&= g\ (p\ \bar{e})\ (f\ (p\ \bar{e})\ (q\ u\ \bar{e})) \\
&= q\ u\ \bar{e}
\end{aligned}$$

Hence $(A, \alpha) : \texttt{Fib}\, \Gamma$. ◀

Note that this proof only uses the fact that $g\, x \circ f\, x = id$ and so in fact the lemma holds more generally in the case where $\langle f, g \rangle$ is just a section-retraction pair rather than a full isomorphism. Although here will we only use it in the context of isomorphisms.

5.1.6 Strictification

▶ **Theorem 5.6.** *There exists a term:*

$$\begin{aligned}
\texttt{strictify} : \{\varphi : \texttt{Cof}\}&(A : [\varphi] \to \mathcal{U})(B : \mathcal{U})(s : (u : [\varphi]) \to (A\, u \cong B)) \to \\
&(B' : \mathcal{U}) \times \{s' : B' \cong B \mid \forall (u : [\varphi]).\, A\, u = B' \wedge s\, u = s'\}
\end{aligned}$$

In words, this says that given any object $B : \mathcal{U}$ and any cofibrant partial object $A : [\varphi] \to \mathcal{U}$ such that A is isomorphic to B everywhere it is defined, then one can can construct a new object $B' : \mathcal{U}$ which extends A, is isomorphic to B, and this isomorphism extends the original isomorphism.

Proof. See [13, Theorem 8.4] for a proof that this property holds in the cubical sets model, and more generally in many other models (classically, in all presheaf models). The construction depends on the fact that, externally, cofibrant propositions are pointwise decidable and so we can simply define (B', s') to be either (A, s) or (B, id) by case analysis on φ at each point. ◀

We now lift this strictification property from objects to fibrations.

▶ **Theorem 5.7.** *Given $\Gamma : \mathcal{U}$ and $\Phi : \Gamma \to \mathtt{Cof}$, a partial fibration $A : \mathtt{Fib}(\Gamma|\Phi)$ and a total fibration $B : \mathtt{Fib}\ \Gamma$ with $iso : A \cong B[\iota]$, we can construct a new type and isomorphism:*

$$A' : \mathtt{Fib}(\Gamma) \qquad \textit{and} \qquad iso' \ : \ A' \cong B$$

such that

$$A'[\iota] = A \qquad \textit{and} \qquad iso' \circ \iota = iso$$

where ι is the inclusion $\Gamma|\Phi \rightarrowtail \Gamma$.

Proof. Given $\Gamma : \mathcal{U}$, $\Phi : \Gamma \to \mathtt{Cof}$, $(A, \alpha) : \mathtt{Fib}(\Gamma|\Phi)$ and $(B, \beta) : \mathtt{Fib}\ \Gamma$ with $iso : A \cong B \circ \iota$, we define A', iso' as:

$$A'\, x \triangleq \mathtt{fst}(\mathtt{strictify}(A(x, \_), B\, x, iso(x, \_)))$$

$$iso'\, x \triangleq \mathtt{snd}(\mathtt{strictify}(A(x, \_), B\, x, iso(x, \_)))$$

Now consider the equalities that are required to hold. From the properties of `strictify` we already have that $A' \circ \iota = A$ and $iso' \circ \iota = iso$. Therefore we just need to define a composition structure $\alpha' : \mathtt{isFib}\ A'$ such that $\alpha'[\iota] = \alpha$.

Since $A' \cong B$ and $\beta : \mathtt{isFib}\ B$ we can use Lemma 5.5 to deduce that A' has a composition structure, which we call α'_{pre}. We then define $\alpha' \triangleq \mathtt{realign}(\Phi, \alpha, \alpha'_{pre})$ using Lemma 5.3. ◀

5.1.7 Misaligned paths between fibrations

We now introduce a new relation between fibrations which we call a misaligned path. This is similar to the notion of path between fibrations introduced in Definition 5.2, except that rather than being equal to A and B at the endpoints, the path only need be isomorphic to A and B at the endpoints.

▶ **Definition 5.8** (Misaligned path equality between fibrations)**.** Define the type of misaligned paths between CCHM fibrations $\_ \sim_{\cong} \_ : \{\Gamma : \mathcal{U}\} \to \mathtt{Fib}\,\Gamma \to \mathtt{Fib}\,\Gamma \to \mathcal{U}_1$ by

$$(A, \alpha) \sim_{\cong} (B, \beta) \triangleq ((P, \rho) : \mathtt{Fib}(\Gamma \times \mathtt{I})) \times (A \cong P \circ \langle id, \mathtt{0}\rangle) \times (B \cong P \circ \langle id, \mathtt{1}\rangle)$$

We can show that every misaligned path can be improved to a regular path between fibrations. First, we introduce a new construction on fibrations.

▶ **Definition 5.9.** Given fibrations $A, B : \mathtt{Fib}\ \Gamma$ we define a new fibration

$$A \veebar B \ : \ \mathtt{Fib}((\Gamma \times \mathtt{I})|\Phi) \qquad \text{where} \qquad \Phi(x, i) \triangleq (i = \mathtt{0}) \vee (i = \mathtt{1})$$

given by $(A, \alpha) \veebar (B, \beta) \triangleq (C, \gamma)$ where

$$C : (\Gamma \times \mathtt{I})|\Phi \to \mathcal{U}$$
$$C\ ((x, i), u) \triangleq ((\lambda\_ : [i = \mathtt{0}] \to A\, x) \cup (\lambda\_ : [i = \mathtt{1}] \to B\, x))\, u$$

Here C is a sort of disjoint union of the families A and B, observing that $(\Gamma \times \mathtt{I})|\Phi \cong \Gamma + \Gamma$ then we can think of C as essentially being $[A, B] : \Gamma + \Gamma \to \mathcal{U}$.

To see that C is fibrant we observe that the interval $\mathtt{I}$ is internally connected in the sense of $\mathtt{ax}_1$ in [13, Figure 1]. This means that any path $p : \mathtt{I} \to (\Gamma \times \mathtt{I})|\Phi$ must either factor as $p = \langle p', \mathtt{0}, * \rangle$ or as $p = \langle p', \mathtt{1}, * \rangle$. Therefore any composition problem for C must lie either entirely in A, in which case we use α to construct a solution, or entirely in B, in which case we use β. For further detail we refer the reader to [13, Theorem 7.3] where the family C occurs as an intermediate construction.

▶ **Definition 5.10.** Given $\Gamma : \mathcal{U}$, $A, B : \mathtt{Fib}\,\Gamma$ and $D : \mathtt{Fib}(\Gamma \times \mathtt{I})$ with isomorphisms $iso_0 : A \cong D[\langle id, \mathtt{0}\rangle]$ and $iso_1 : B \cong D[\langle id, \mathtt{1}\rangle]$ then define $iso_0 \veebar iso_1 : A \veebar B \cong D[\iota]$ as follows. Given $(x, i, u) : (\Gamma \times \mathtt{I})|\Phi$:

$$(iso_0 \veebar iso_1)\,(x, i, u) : (\mathtt{fst}(A \veebar B))\;(x, i, u) \to (\mathtt{fst}\,D)\;(x, i)$$

$$(iso_0 \veebar iso_1)\,(x, i, u) \triangleq \begin{cases} iso_0\,x & \text{when } u : [i = \mathtt{0}] \\ iso_1\,x & \text{when } u : [i = \mathtt{1}] \end{cases}$$

Observe that for all $A, B : \mathtt{Fib}\,\Gamma$ we have $(A \veebar B)[\langle id, \mathtt{0}, * \rangle] = A$ and $(A \veebar B)[\langle id, \mathtt{1}, * \rangle] = B$, and for all $iso_0 : A \cong D[\langle id, \mathtt{0}\rangle]$ and $iso_1 : B \cong D[\langle id, \mathtt{1}\rangle]$ we have $(iso_0 \veebar iso_1) \circ \langle id, \mathtt{0}, * \rangle = iso_0$ and $(iso_0 \veebar iso_1) \circ \langle id, \mathtt{1}, * \rangle = iso_1$. We now use this construct to show the following result:

▶ **Lemma 5.11.** *There exists a function*

$$\mathtt{improve} : \{\Gamma : \mathcal{U}\}\{A\;B : \mathtt{Fib}\,\Gamma\} \to A \sim_{\cong} B \to A \sim_{\mathcal{U}} B$$

Proof. Take $\Gamma : \mathcal{U}$, $A, B : \mathtt{Fib}\,\Gamma$ and $(P, iso_0, iso_1) : A \sim_{\cong} B$ and observe that $iso_0 \veebar iso_1 : A \veebar B \cong P[\iota]$. Therefore we can use Theorem 5.7 to strictify P in order to get $P' : \mathtt{Fib}(\Gamma \times \mathtt{I})$ such that $P'[\iota] = A \veebar B$, where ι is the restriction $(\Gamma \times \mathtt{I})|\Phi \to \Gamma \times \mathtt{I}$. Now consider reindexing P' along $\langle id, \mathtt{0}\rangle : \Gamma \to \Gamma \times \mathtt{I}$ we get:

$$P'[\langle id, \mathtt{0}\rangle] = P'[\iota \circ \langle id, \mathtt{0}, * \rangle] = P'[\iota][\langle id, \mathtt{0}, * \rangle] = (A \veebar B)[\langle id, \mathtt{0}, * \rangle] = A$$

and similarly $P'[\langle id, \mathtt{1}\rangle] = B$. Therefore we have $P' : A \sim_{\mathcal{U}} B$ as required. ◀

5.1.8 Function extensionality

As discussed previously, function extensionality holds straightforwardly in any type theory which includes an interval object/type with certain computational properties, cf. [15, Lemma 6.3.2]. See [13, Remark 5.16] and [8, Section 3.2] for a proof in the case of cubical type theory.

5.1.9 Axioms (1), (2), (4) and (5)

As discussed previously, we can satisfy axioms (1) and (2) by showing that there is a way to construct paths between strictly isomorphic (fibrant) types $A, B : \mathtt{Fib}\,\Gamma$.

▶ **Theorem 5.12.** *Given fibrations $A, B : \mathtt{Fib}\,\Gamma$ with $iso : A \cong B$ we can construct a path $\mathtt{isopath}(iso) : A \sim_{\mathcal{U}} B$.*

Proof. Given A, B, f, g as above, let $B' \triangleq B[\mathtt{fst}] : \mathtt{Fib}(\Gamma \times \mathtt{I})$ and note that $iso : A \cong B'[\langle id, \mathtt{0}\rangle]$ and $id : B \cong B'[\langle id, \mathtt{1}\rangle]$ where id is the obvious isomorphism $B \cong B$. Therefore we can define

$$\mathtt{isopath}(iso) \triangleq \mathtt{improve}(B[\mathtt{fst}], iso, id) : A \sim_{\mathcal{U}} B$$

as required. Note that, in this case, $\mathtt{improve}$ will in fact only improve $B[\mathtt{fst}]$ at $\mathtt{0}$, since at $\mathtt{1}$ we improve along the identity, which does nothing. ◀

▶ **Corollary 5.13.** *Axioms (1) and (2) hold in the cubical sets model.*

Proof. The obvious isomorphisms $A \cong A \times 1$ and $\sum_{a:A} \sum_{b:B} C\ a\ b \cong \sum_{b:B} \sum_{a:A} C\ a\ b$ are both clearly strict isomorphisms in the sense of Definition 5.4. Therefore we can construct the required paths $A \sim_{\mathcal{U}} (A \times 1)$ and $(\sum_{a:A} \sum_{b:B} C\ a\ b) \sim_{\mathcal{U}} (\sum_{b:B} \sum_{a:A} C\ a\ b)$. Hence axioms (1) and (2) hold.

Note that in order interpret axioms (1) and (2) using `isopath` we need to know that `isopath` is stable under reindexing (substitution in the type theory). This will be the case because most of the constructions used to define it (strictification, closure under isomorphism, etc) are all performed fiberwise and hence will be stable under reindexing. The only exception is the realignment lemma, which redefines the entire composition structure. However, we previously showed `realign` to be stable under reindexing. Therefore `isopath` will also be stable under reindexing. ◀

We have seen that we can easily satisfy axioms (1) and (2) in the cubical sets model. However, we also need to know what happens when we coerce along these equalities. This can be stated in general for any strictly isomorphic types.

▶ **Theorem 5.14.** *Given fibrations $(A, \alpha), (B, \beta) : \mathtt{Fib}\ \Gamma$ with $\langle f, g \rangle : A \cong B$, coercing along $\mathtt{isopath}(\langle f, g \rangle)$ is (propositionally) equal to applying f.*

Proof. Take $(A, \alpha), (B, \beta), f, g$ as above and let $(P, \rho) = \mathtt{isopath}(\langle f, g \rangle)$. By unfolding the constructions used we can see that ρ was obtained by realigning some ρ_{pre}, which in turn was obtained by transferring $\beta[\mathtt{fst}]$ across the isomorphism:

$$iso'\,(x, i) = \mathtt{snd}(\mathtt{strictify}((A, \beta) \veebar (B, \beta)(x, i, \_), B\,x, (\langle f, g \rangle \veebar id)\,(x, i, \_))) : P\,x \cong B\,x$$

Now consider arbitrary $x : \Gamma, a_0 : A\ x$ and note that

$$iso'\,(x, \mathtt{0}) = (\langle f, g \rangle \veebar id)\,(x, \mathtt{0}) = \langle f, g \rangle\,x = (f\,x, g\,x)$$

and

$$iso'\,(x, \mathtt{1}) = (\langle f, g \rangle \veebar id)\,(x, \mathtt{1}) = (id, id)$$

Now calculate:

$$\begin{aligned}
&\mathtt{coerce}\ \mathtt{isopath}(\langle f, g \rangle)\ x\ a_0 \\
&\quad = \rho\ \mathtt{0}\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ a_0 && \text{by unfolding definitions}^2 \\
&\quad = \rho_{pre}\ \mathtt{0}\ \langle x, id \rangle\ (\forall i.(i = \mathtt{0} \vee i = \mathtt{1}))\ q\ a_0 && \text{by Lemma 5.3, for some } q \\
&\quad = \rho_{pre}\ \mathtt{0}\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ a_0 && \text{by definition of } \forall \\
&\quad = \mathtt{snd}(iso'(x, \mathtt{1}))\ (\beta\ \mathtt{0}\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ (\mathtt{fst}(iso'(x, \mathtt{0}))\ a_0)) && \text{by Lemma 5.5} \\
&\quad = \beta\ \mathtt{0}\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ (\mathtt{fst}(iso'(x, \mathtt{0}))\ a_0)) && \text{since } \mathtt{snd}(iso'(x, \mathtt{1})) = id \\
&\quad = \beta\ \mathtt{0}\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ (f\ x\ a_0) && \text{since } \mathtt{fst}(iso'(x, \mathtt{0})) = f\ x
\end{aligned}$$

Since this is merely a trivial/empty composition applied to $f\ x\ a_0$ we can construct a path from $f\ x\ a_0$ to $\mathtt{coerce}\ \mathtt{isopath}(\langle f, g \rangle)\ x\ a_0$ like so:

$$\mathtt{fill}\ \mathtt{0}\ \beta\ \langle x, id \rangle\ \bot\ \mathtt{elim}_\emptyset\ (f\ x\ a_0) : f\ x\ a_0 \sim \mathtt{coerce}\ \mathtt{isopath}(\langle f, g \rangle)\ x\ a_0$$

Therefore, coercing along $isopath(\langle f, g \rangle)$ is always propositionally equal to applying f. ◀

▶ **Corollary 5.15.** *Axioms (4) and (5) hold in the cubical sets model (for the terms constructed in Corollary 5.13).*

Proof. By Theorem 5.14. ◀

5.1.10 Axiom (3)

In light of the previous section, the only axiom remaining is axiom (3). Our goal here is, given a contractible fibration $A : \texttt{Fib}\ \Gamma$, to define a path $A \sim_{\mathcal{U}} 1$. Note that, for any $\Gamma : \mathcal{U}$, there exists a unique fibration structure $!_1$ such that $(\lambda\_ \to 1, !_1) : \texttt{Fib}(\Gamma)$. Therefore we will ambiguously write $1 : \texttt{Fib}(\Gamma)$ for the pair $(\lambda\_ \to 1, !_1)$.

▶ **Definition 5.16** (The contraction of a family)**.** Given a family $A : \Gamma \to \mathcal{U}$ we define the contraction of A as

$$C_A : \Gamma \times \mathtt{I} \to \mathcal{U}$$
$$C_A(x, i) \triangleq [i = \mathtt{0}] \to A(x)$$

We now need to show that C_A is fibrant when A is both fibrant and contractible. First, we restate the property of being contractible (Definition 2.1) in the internal type theory.

▶ **Definition 5.17.** A type A is said to be *contractible* if it has a centre of contraction $a_0 : A$ and every element $a : A$ is propositionally equal to a_0, that is, there exists a path $a_0 \sim a$. Therefore a type is contractible if $\texttt{Contr}\,A$ is inhabited, where $\texttt{Contr} : \mathcal{U} \to \mathcal{U}$ is defined by

$$\texttt{Contr}\,A \triangleq (a_0 : A) \times ((a : A) \to a_0 \sim a)$$

We say that a family $A : \Gamma \to \mathcal{U}$ is contractible if each of its fibres is and abusively write

$$\texttt{Contr}\,A \triangleq (x : \Gamma) \to \texttt{Contr}(A\,x)$$

Next we recall the notion of an extension structure [13, Definition 6.4].

▶ **Definition 5.18** (Extension structures)**.** The type of extension structures, $\texttt{Ext} : \mathcal{U} \to \mathcal{U}$, is given by

$$\texttt{Ext}\,A \triangleq (\varphi : \texttt{Cof})(f : [\varphi] \to A) \to \{a : A \mid (\varphi, f) \nearrow a\}$$

Having an extension structure for a type $A : \mathcal{U}$ allows us to extend any partial element of A to a total element. As before we say that a family $A : \Gamma \to \mathcal{U}$ has an extension structure if each of its fibres do, and write

$$\texttt{Ext}\,A \triangleq (x : \Gamma) \to \texttt{Ext}(A\,x)$$

▶ **Lemma 5.19.** *Any family $A : \Gamma \to \mathcal{U}$ that is both fibrant and contractible is also extendable in the sense of Defintion 5.18.*

Proof. By [13, Lemma 6.6]. ◀

[2] Note that there are different ways to interpret `coerce` in the model. This interpretation is not in general the same as the one obtained by directly interpreting Definition 3.1. However, the two interpretations will always be path equal in the model (the other interpretation will have more trivial/empty compositions), and so the result still holds when using the other interpretation.

Now we can construct a fibrancy structure for C_A as follows:

▶ **Theorem 5.20.** *If* $(A, \alpha) : \texttt{Fib}\ \Gamma$ *is contractible then we can construct a composition structure for* C_A*.*

Proof. Take $(A, \alpha) : \texttt{Fib}\ \Gamma$ as above. Since A is both fibrant and contractible we can construct an extension structure $\epsilon : \texttt{Ext}\ A$. We can then define a composition structure $c_\alpha : \texttt{isFib}(C_A)$ as follows, given

$$e : \texttt{\{0,1\}} \qquad p : \texttt{I} \to \Gamma \times \texttt{I} \qquad \varphi : \texttt{Cof} \qquad f : [\varphi] \to \Pi_{\texttt{I}}(C_A \circ p) \qquad c_0 : C_A\,(p\,e)$$

we must define as term of type $C_A\,(p\,\bar{e})$. Since this type is defined to be $[\texttt{snd}(p\,\bar{e}) = \texttt{0}] \to A$, we define the composition like so:

$$(c_\alpha\ e\ p\ \varphi\ f\ c_0)\ u \triangleq \epsilon\ (p\,\bar{e})\ \varphi\ (\lambda v \to f\,v\,\bar{e}\,u)$$

Given $v : [\varphi]$ we have:

$$c_\alpha\ e\ p\ \varphi\ f\ c_0 = \lambda u \to \epsilon\ (p\,\bar{e})\ \varphi\ (\lambda v \to f\,v\,\bar{e}\,u) = \lambda u \to f\,v\,\bar{e}\,u = f\,v\,\bar{e}$$

as required. Therefore we have a defined a valid composition operation for C_A. ◀

▶ **Theorem 5.21.** *There exists a function*

$$\texttt{contract} : \{\Gamma : \mathcal{U}\}(A : \texttt{Fib}\ \Gamma) \to \texttt{Contr}\ \ A \to A \sim_{\mathcal{U}} 1$$

Proof. Given $\Gamma : \mathcal{U}$, $(A, \alpha) : \texttt{Fib}\ \Gamma$ and $\epsilon : \texttt{Contr}\ \ A$, we obverse that

$$C_A[\langle id, \texttt{0}\rangle](x) = C_A(x, \texttt{0}) = [\texttt{0} = \texttt{0}] \to A(x) \cong 1 \to A(x) \cong A(x)$$

and

$$C_A[\langle id, \texttt{1}\rangle](x) = C_A(x, \texttt{1}) = [\texttt{1} = \texttt{0}] \to A(x) \cong \emptyset \to A(x) \cong 1$$

Therefore we have $((C_A, c_\alpha), iso_A, iso_1) : A \sim_{\cong} 1$ where $iso_A : A \cong C_A[\langle id, \texttt{0}\rangle]$ and $iso_1 : 1 \cong C_A[\langle id, \texttt{1}\rangle]$ are the obvious isomorphisms indicated above. Hence we can define

$$\texttt{contract}((A, \alpha), \epsilon) \triangleq \texttt{improve}((C_A, c_\alpha), iso_A, iso_1) : (A, \alpha) \sim_{\mathcal{U}} 1$$

as required. ◀

▶ **Corollary 5.22.** *Cubical type theory with the cubical sets model supports axiom (3).*

As in Corollary 5.13 we need to check that `contract` is stable under reindexing (substitution). This holds for the same reasons as before, namely that the only non fibrewise construction used in the definition of `contract` is `realign` which we previously showed to be stable under reindexing.

6 An application to an open problem in type theory

In Section 2 we defined *funext* to be the principle which says that two functions $f, g : \prod_{x:A} B(x)$ are equal if they are pointwise equal: $f \approx g \triangleq \prod_{x:A} f(x) = g(x)$. That is, we assumed the existence of a term:

$$funext_{i,j} : \prod_{A:U_i}\ \prod_{B:A \to U_j}\ \prod_{f,g:\Pi_{x:A}B(x)} f \approx g \to f = g$$

for all universe levels i, j. This is similar to the statement of naive univalence, UA, from Definition 3.3 and we call this principle naive function extensionality.

As with proper univalence (Definition 3.2), we could have instead stated that the canonical map $happly : (f = g) \to f \approx g$ is an equivalence. In fact, these two formulations turn out to be equivalent.

▶ **Theorem 6.1** (due to Voevodsky). *Naive function extensionality is logically equivalent to the proper function extensionality axiom. That is, the existence of a term:*

$$funext_{i,j} : \prod_{A:U_i} \prod_{B:A \to U_j} \prod_{f,g:\Pi_{x:A} B(x)} f \approx g \to f = g$$

is logically equivalent to the statement that, for all types $A : U_i$, $B : A \to U_j$ and maps $f, g : \prod_{x:A} B(x)$, the map $happly : (f = g) \to (f \approx g)$ is an equivalence.

Proof. For the forwards direction: assuming *funext* as above, it is easy to derive a proof of *weak function extensionality* [15, Definition 4.9.1]. This in turn implies the proper function extensionality axiom by [15, Theorem 4.9.5]. The reverse direction follows trivially. ◀

Compare this result with Theorem 3.5 where we saw that naive univalence with a computation rule is logically equivalent to the proper univalence axiom. In the case of function extensionality we did not need to assume any sort of computation rule about *funext*. Therefore an obvious question is whether this computation rule is in fact necessary in the case of univalence, or whether, as is the case with function extensionality, it is in fact redundant.

▶ **Conjecture 6.2.** *Naive univalence implies the proper univalence axiom. That is, given UA_i, it follows that for all types $A, B : U_i$ the map $idtoeqv : (A = B) \to (A \simeq B)$ is an equivalence.*

To the authors' best knowledge the status of Conjecture 6.2 is currently unknown. It is certainly not inconsistent since there are models where naive univalence fails to hold, such as the *Set*-valued model [10], and models where full univalence holds, such as the cubical sets model [8]. However it is not clear whether Conjecture 6.2 is either a theorem of type theory, cf. the case with function extensionality, or whether there are models which validate UA but which do not validate the proper univalence axiom.

The work presented here may offer an approach to tackling this problem, by reducing it to the following:

▶ **Conjecture 6.3.** *Axioms (1)-(3) imply axioms (4)-(5), for possibly modified unit and flip. That is, if for all $A, B : U_i$, $C : A \to B \to U_i$ we have:*

$$A = \sum_{a:A} 1 \qquad \sum_{a:A} \sum_{b:B} C\ a\ b = \sum_{b:B} \sum_{a:A} C\ a\ b \qquad isContr(A) \to A = 1$$

then there exist terms unit and flip, with types as in Table 1, for which the following equalities hold:

$$\texttt{coerce}\ unit\ a = (a, *) \qquad \texttt{coerce}\ flip\ (a, b, c) = (b, a, c)$$

for all $a : A$, $b : B$ and $c : C\,a\,b$.

▶ **Theorem 6.4.** *In the presence of function extensionality, Conjecture 6.2 and Conjecture 6.3 are logically equivalent.*

Proof. For the forwards direction, assume function extensionality, 6.2 and axioms (1)-(3). By Theorem 4.1 we deduce that naive univalence, UA_i, holds. Therefore by our assumption of 6.2 we deduce the proper univalence axiom for U_i. Hence, by Corollary 4.4, we deduce axioms (1)-(5) (possibly with different proof terms than our existing assumptions of axioms (1)-(3)). Therefore the conclusion of 6.3 holds.

For the reverse direction, assume function extensionality, 6.3 and naive univalence. By Theorem 4.1 we deduce that axioms (1)-(3) hold. Therefore by our assumption of 6.3 we deduce axioms (4)-(5) also hold. Hence, by Corollary 4.4, we deduce the proper univalence axiom. ◀

This result may be useful in tackling the open question of whether Conjecture 6.2 is a theorem of type theory, or whether there are in fact models in which it does not hold. This is because finding models where the conclusions of Conjecture 6.3 do not hold given the assumptions, or showing that no such models exist, seems an easier task. For example, consider the case where the first conclusion fails, that is, where $\texttt{coerce}\, \mathit{unit}\, a \neq (a, *)$ for some $A : U$ and $a : A$. If this is the case then we have $\texttt{fst} \circ (\texttt{coerce}\ \mathit{unit}) : \{A : U\} \to A \to A$ which is not equal to the identity function. We note that the existence of such a term has interesting consequences relating to parametricity and excluded middle [7], and potentially informs our search about the type of models which might invalidate Conjecture 6.2. However, we leave further investigation of this problem to future work.

References

1 Agda Project. URL: `http://wiki.portal.chalmers.se/agda`.

2 C. Angiuli, G. Brunerie, T. Coquand, K.-B. Hou (Favonia), R. Harper, and D. R. Licata. Cartesian Cubical Type Theory (preprint), 2017. URL: `https://github.com/dlicata335/cart-cube/blob/master/cart-cube.pdf`.

3 S. Awodey. A cubical model of homotopy type theory. *arXiv preprint arXiv:1607.06413*, 2016. URL: `https://arxiv.org/abs/1607.06413`.

4 R. Balbes and P. Dwinger. *Distributive Lattices.* University of Missouri Press, 1975.

5 M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26, pages 107–128, 2014.

6 L. Birkedal, A. Bizjak, R. Clouston, H. B. Grathwohl, B. Spitters, and A. Vezzosi. Guarded Cubical Type Theory: Path Equality for Guarded Recursion. In *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 23:1–23:17, 2016.

7 A. B. Booij, M. H. Escardó, P. L. Lumsdaine, and M. Shulman. Parametricity, automorphisms of the universe, and excluded middle. *arXiv preprint arXiv:1701.05617*, 2017. URL: `https://arxiv.org/abs/1701.05617`.

8 C. Cohen, T. Coquand, S. Huber, and A. Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In *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, 2018.

9 P. Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 1996.

10 M. Hofmann. Syntax and Semantics of Dependent Types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

11 M. Hofmann and T. Streicher. Lifting Grothendieck Universes (Unpublished note), 1997. URL: `https://www2.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf`.

12 M. E. Maietti. Modular Correspondence between Dependent Type Theories and Categories including Pretopoi and Topoi. *Mathematical Structures in Computer Science*, 15:1089–1149, 2005.

13 I. Orton and A. M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science*, 2018. Special issue for CSL 2016, to appear. URL: `https://arxiv.org/abs/1712.04864`.

14 A. M. Pitts. Nominal Presentation of Cubical Sets Models of Type Theory. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 39. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

15 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. Univalent Foundations Project, Institute for Advanced Study, 2013. URL: `http://homotopytypetheory.org/book`.

On Equality of Objects in Categories in Constructive Type Theory

Erik Palmgren
Department of Mathematics, Stockholm University, Stockholm, Sweden
palmgren@math.su.se

Abstract

In this note we remark on the problem of equality of objects in categories formalized in Martin-Löf's constructive type theory. A standard notion of category in this system is E-category, where no such equality is specified. The main observation here is that there is no general extension of E-categories to categories with equality on objects, unless the principle Uniqueness of Identity Proofs (UIP) holds. We also introduce the notion of an H-category with equality on objects, which makes it easy to compare to the notion of univalent category proposed for Univalent Type Theory by Ahrens, Kapulkin and Shulman.

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases type theory, formalization, category theory, setoids

Digital Object Identifier 10.4230/LIPIcs.TYPES.2017.7

Acknowledgements The author is grateful to the referees for spotting a false theorem in a previous version of this paper.

1 Introduction

In this note we remark on the problem of equality of objects in categories formalized in Martin-Löf's constructive type theory. A common notion of category in this system is E-category [1], where no such equality is specified. The main observation here is that there is no general extension of E-categories to categories with equality on objects, unless the principle Uniqueness of Identity Proofs (UIP) holds. In fact, for every type A, there is an E-groupoid A^ι which cannot be so extended. We also introduce the notion of an H-category, a variant of category, which makes it easy to compare to the notion of *univalent category* proposed in Univalent Type Theory [9].

When formalizing mathematical structures in constructive type theory it is common to interpret the notion of set as a type together with an equivalence relation, and the notion of function between sets as a function or operation that preserves the equivalence relations. Such functions are called *extensional functions*. This way of interpreting sets was adopted in Bishop's seminal book [4] on constructive analysis from 1967. In type theory literature [3, 6, 8, 10] such sets are called *setoids*. Formally a setoid $X = (|X|, =_X, \mathrm{eq}_X)$ consists of a type $|X|$ together with a binary relation $=_X$, and a proof object eq_X witnessing $=_X$ being an equivalence relation. We usually suppress the proof object. An extensional function between setoids $f : X \to Y$ consists of a type-theoretic function $|f| : |X| \to |Y|$, and a proof that f respects the equivalence relations, i.e. $|f|(x) =_Y |f|(u)$ whenever $x =_X u$. One writes $x : X$ for $x : |X|$, and $f(x)$ for $|f|(x)$ to simplify notation. Every type A comes with a minimal equivalence relation $\mathrm{I}_A(\cdot,\cdot)$, the so-called identity type for A. We sometimes write $a \doteq b$ for $\mathrm{I}_A(a, b)$, when the type can be inferred. The principle of Uniqueness of Identity Proofs (UIP) for a type A states that

$$(\mathrm{UIP}_A) \qquad (\forall a, b : A)(\forall p, q : a \doteq b) p \doteq q$$

(using the propostions-as-types convention that $\forall$ is Π, $\exists$ is Σ etc.) This principle is not assumed in basic type theory, but can be proved for types A where $\mathrm{I}_A(\cdot, \cdot)$ is a decidable relation (Hedberg's Theorem [9]). Another essential notion used in this paper is that of family of setoids indexed by a setoid. There are several choices that can be made but the one corresponding to fibers $\{f^{-1}(a)\}_{a \in A}$ of an extensional function $f : B \to A$ between setoids is the notion of a proof-irrelevant family. Let A be a setoid. A *proof-irrelevant family B of setoids over A,* assigns to each $a : |A|$, a setoid $B(a) = (|B(a)|, =_{B(a)}, eq_{B(a)})$, and to each proof object $p : a =_A b$ an extensional function $B(p) : B(a) \to B(b)$ (the *transport map* associated with p). The transport maps should satisfy the following conditions

- $B(p)(x) =_{B(a)} x$ for all $x : B(a)$ and $p : a =_A a$ (identity)
- $B(p)(x) =_{B(b)} B(q)(x)$ for all $x : B(a)$ and $p, q : a =_A b$ (proof-irrelevance)
- $B(q)(B(p)(x)) =_{B(c)} B(r)(x)$ for all $x : B(a)$ and $p : a =_A b$, $q : b =_A c$, $r : a =_A c$ (functoriality)

From these conditions follows easily that each $B(p)$ is an isomorphism which is independent of the proof object p. Hence *proof-irrelevance.* (An equivalent definition is obtained by considering A as a discrete E-category $A^\#$ (whose objects are elements of $|A|$ and whose hom-setoids are $\mathrm{Hom}(a, b) = (a =_A b, \sim)$ with $p \sim q$ always true) and B as a functor from this category to the E-category of setoids. This uses concepts only defined below.)

In Univalent Type Theory [9] the identity type is axiomatized so as to allow quotients, and many other constructions. This makes it possible to avoid the extra complexity of setoids and their defined equivalence relations.

These two approaches to type theory, may lead to different developments of category theory. In both cases there are notions of categories, *E-categories* and *precategories*, which are incomplete in some sense.

2 Categories in standard type theory

Categories [5] are commonly formalized in set theory in two ways, one is the essentially algebraic formulation, where objects, arrows, and composable arrows each form sets (or classes), with appropriate operations, and the other one is via objects and hom-sets (hom-classes). Set theory gives automatically a notion of equality on objects imposed by the equality of the theory. These definitions can be carried over to type theory and setoids, by taking care to make all constructions extensional.

In type theory, an *essentially algebraically presented category*, or *EA-category* for short, is formulated as follows. It consists of three setoids $\mathrm{Ob}(\mathcal{C})$, $\mathrm{Arr}(\mathcal{C})$ and $\mathrm{Cmp}(\mathcal{C})$ of objects, arrows and composable pairs of arrows, respectively. Objects are thus supposed to be equipped with equality. There are extensional functions, providing identity arrows to objects, $1 : \mathrm{Ob} \to \mathrm{Arr}$, providing domains and codomains to arrows $\mathrm{dom}, \mathrm{cod} : \mathrm{Arr} \to \mathrm{Ob}$, a composition function $\mathrm{cmp} : \mathrm{Cmp} \to \mathrm{Arr}$, and selection functions $\mathrm{fst}, \mathrm{snd} : \mathrm{Cmp} \to \mathrm{Arr}$ satisfying familiar equations, with the axiom that for a pair of arrows f, g:

$$\mathrm{cod}(g) = \mathrm{dom}(f) \iff (\exists u : \mathrm{Cmp})\, g = \mathrm{fst}(u) \wedge f = \mathrm{snd}(u)$$

In this case $\mathrm{cmp}(u)$ will be the composition $f \circ g$. See [5, 8] for axioms and details.

The hom-set formulation in type theory is the following [7]: A *hom-family presented category $\mathcal{C}$*, or just *HF-category*, consists of a setoid C of objects, and a (proof irrelevant) setoid family of homomorphisms Hom indexed by the product setoid $C \times C$. We have $1_a : \mathrm{Hom}(a,a)$, and an extensional composition $\circ_{a,b,c} : \mathrm{Hom}(b,c) \times \mathrm{Hom}(a,b) \to \mathrm{Hom}(a,c)$ satisfying

- $f \circ_{a,a,b} 1_a =_{\mathrm{Hom}(a,b)} f \quad 1_b \circ_{a,b,b} f = f$, if $f : \mathrm{Hom}(a,b)$,
- $f \circ_{a,c,d} (g \circ_{a,b,c} h) =_{\mathrm{Hom}(a,d)} (f \circ_{b,c,d} g) \circ_{a,b,d} h$, if $f : \mathrm{Hom}(c,d)$, $g : \mathrm{Hom}(b,c)$, $h : \mathrm{Hom}(a,b)$.

For $p : a =_C c$ and $q : b =_C d$, the transport map goes as follows

$$\mathrm{Hom}(p,q) : \mathrm{Hom}(a,b) \to \mathrm{Hom}(c,d).$$

The transport maps have to satisfy the following coherence conditions:

- $\mathrm{Hom}(p,p)(1_a) =_{\mathrm{Hom}(a',a')} 1_{a'}$ for $p : a =_C a'$
- $\mathrm{Hom}(p,r)(f \circ_{a,b,c} g) =_{\mathrm{Hom}(a',c')} \mathrm{Hom}(q,r)(f) \circ_{a',b',c'} \mathrm{Hom}(p,q)(g)$ for $p : a =_C a'$, $q : b =_C b'$, $r : c =_C c'$, $f : \mathrm{Hom}(b,c)$ and $g : \mathrm{Hom}(a,b)$.

▶ Remark. The coherence conditions can be captured more briefly by just stating that 1 and $\circ$ are elements in the following dependent product setoids

(a) $1 : \Pi(C, \mathrm{Hom}\langle \mathrm{id}_C, \mathrm{id}_C \rangle)$

(b) $\circ : \Pi(C^3, \mathrm{Hom}\langle \pi_2, \pi_3 \rangle \times \mathrm{Hom}\langle \pi_1, \pi_2 \rangle \to \mathrm{Hom}\langle \pi_1, \pi_3 \rangle)$.

In more detail, the product setoids in (a) and (b) are made using the following constructions:

Let $\mathrm{Fam}(A)$ denote the type of proof irrelevant families over the setoid A. Such families are closed under the following pointwise operations:

If $F, G : \mathrm{Fam}(A)$, then $F \times G : \mathrm{Fam}(A)$ and $F \to G : \mathrm{Fam}(A)$.

If $F : \mathrm{Fam}(A)$, and $f : B \to A$ is extensional, then the composition $Ff : \mathrm{Fam}(B)$.

The cartesian product $\Pi(A, F)$ of a family $F : \mathrm{Fam}(A)$ consists of pairs $f = (|f|, \mathrm{ext}_f)$ where $f : (\Pi x : |A|)|F(x)|$ and ext_f is a proof object that witnesses that $|f|$ is extensional, that is

$$\mathrm{ext}_f : (\forall x, y : A)(\forall p : x =_A y)[F(p)(|f|(x)) =_{F(y)} |f|(y)].$$

Two such pairs f and f' are extensionally equally if and only if $|f|(x) =_{F(x)} |f'|(x)$ for all $x : A$. Then it is straightforward to check that $\Pi(A, F)$ is a setoid.

3 E-categories and H-categories in standard type theory

According to the philosophy of category theory, truly categorical notions should not refer to equality of objects. This has a very natural realization in type theory, since there, unlike in set theory, we can choose *not to impose* an equality on a type. This leads to the notion of *E-category* [1], which is essentially an HF-category with equality on objects taken away, and the corresponding transport maps removed.

An *E-category* $\mathcal{C} = (C, \mathrm{Hom}, \circ, 1)$ is the formulation of a category where there is a *type* C of objects, but no imposed equality, and for each pair of objects a, b there is a setoid $\mathrm{Hom}(a,b)$ of morphisms from a to b. The composition is an extensional function

$$\circ : \mathrm{Hom}(b,c) \times \mathrm{Hom}(a,b) \to \mathrm{Hom}_{\mathcal{C}}(a,c).$$

satisfying the familiar laws of associativity and identity. A functor or an *E-functor* between E-categories is defined as usual, but the object part does not need to respect any equality of objects (because there is none).

Now an interesting question is whether we can impose an equality of objects onto an E-category which is compatible with composition, so as to obtain an HF-category? We may consider an intermediate structure on E-categories as follows.

Define an *H-category* $\mathcal{C} = (C, =_C, \mathrm{Hom}, \circ, 1, \tau)$ to be an E-category with an equivalence relation $=_C$ on the objects C, and a family of isomorphisms $\tau_{a,b,p} \in \mathrm{Hom}(a, b)$, for each proof $p : a =_C b$. The morphisms should satisfy the conditions

(H1) $\tau_{a,a,p} = 1_a$ for any $p : a =_C a$

(H2) $\tau_{a,b,p} = \tau_{a,b,q}$ for any $p, q : a =_C b$

(H3) $\tau_{b,c,q} \circ \tau_{a,b,p} = \tau_{a,c,r}$ for any $p : a =_C b$, $q : b =_C c$ and $r : a =_C c$.

Axioms (H1) and (H3) can be replaced by the special cases $\tau_{a,a,\mathrm{ref}(a)} = 1_a$, and $\tau_{b,c,q} \circ \tau_{a,b,p} = \tau_{a,c,\mathrm{tr}(q,p)}$ where ref and tr are specific proofs of reflexivity and transitivity. Note that by these axioms, it follows that each $\tau_{a,b,p}$ is indeed an isomorphism.

A functor between H-categories $\mathcal{C} = (C, =_C, \mathrm{Hom}, \circ, 1, \tau)$ and $\mathcal{D} = (D, =_D \mathrm{Hom}', \circ', 1', \sigma)$ is an E-functor F from $(C, \mathrm{Hom}, \circ, 1)$ to $(D, \mathrm{Hom}', \circ', 1')$ such that $a =_C b$ implies $F(a) =_D F(b)$ and $F(\tau_{a,b,p}) = \sigma_{F(a),F(b),q}$ for $p : a =_C b$ and $q : F(a) =_D F(b)$.

An H-category $\mathcal{C}$ is called *skeletal* if $a =_C b$ whenever a and b are isomorphic in $\mathcal{C}$.

To pass between H- and HF-categories we proceed as follows:
For an H-category $\mathcal{C} = (C, =_C, \mathrm{Hom}, \circ, 1, \tau)$, define a transportation function

$$\mathrm{Hom}(p, q) : \mathrm{Hom}(a, b) \to \mathrm{Hom}(a', b')$$

for $p : a =_C a'$ and $q : b =_C b'$, by

$$\mathrm{Hom}(p, q)(f) = \tau_{b,b',q} \circ f \circ \tau_{a',a,p^{-1}}.$$

It is straightforward to check that this defines an HF-category.

Conversely, an HF-category $\mathcal{C} = (C, \mathrm{Hom}, \circ, 1)$ yields an E-category $(|C|, \mathrm{Hom}, \circ, 1)$ and we can define, an H-structure on it by, for $p : a =_C b$,

$$\tau_{a,b,p} = \mathrm{Hom}(r(a), p)(1_a) : \mathrm{Hom}(a, b).$$

These constructions are inverses to each other, though they do not form an equivalence, since the two categories have different notions of functors.

4 E-categories are proper generalizations of H-categories

The existence of some H-structure on any E-category turns out to be equivalent to UIP.

► **Theorem 1.** *If UIP holds for the type C, then any E-category with objects C can be extended to an H-category.*

Proof. The equivalence relation on C will be $\mathrm{I}_C(\cdot, \cdot)$. Using induction on identity one defines $\tau_{a,b,p} \in \mathrm{Hom}(a, b)$ for $p \in \mathrm{I}(C, a, b)$ by

$$\tau_{a,a,\mathrm{ref}(a)} =_{\mathrm{def}} \mathrm{id}_a.$$

The UIP property implies (H2). Property (H3) follows from transitivity and (H2). ◄

► Remark. We recall that by Hedberg's theorem, UIP holds for a type C, whenever $\mathrm{I}_C(x, y) \vee \neg \mathrm{I}_C(x, y)$, for all $x, y : C$. This explains why the extension problem is trivial in a classical setting.

Let A be an arbitrary type. Define the E-category A^ι where A is the type of objects, and hom setoids are given by

$$\mathrm{Hom}(a,b) =_{\mathrm{def}} (\mathrm{I}_A(a,b), \approx)$$

where $p \approx q$ holds if and only if $\mathrm{I}_{\mathrm{I}_A(a,b)}(p,q)$ is inhabited. Let composition be given by the proof object transitivity, and the identity on a is $\mathrm{ref}(a)$. Then it is well-known that A^ι is an E-groupoid.

▶ **Theorem 2.** *Let A be a type. Suppose that the E-category A^ι can be extended to an H-category. Then UIP holds for A.*

Proof. Suppose that $=_A, \tau$ is an H-structure on A^ι.

Now since $\mathrm{I}_A(a,b)$ is the minimal equivalence relation on A, there is a proof object $f(p) : a =_A b$ for each $p : \mathrm{I}_A(a,b)$. Thus $\tau_{a,b,f(p)} : \mathrm{Hom}(a,b) = \mathrm{I}_A(a,b)$. Let $D(a,b,p)$ be the proposition

$$\tau_{a,b,f(p)} \approx p. \tag{1}$$

By (H1) it holds that

$$\tau_{a,a,f(\mathrm{ref}(a))} \approx \mathrm{ref}(a),$$

i.e. $D(a,a,\mathrm{ref}(a))$. Hence by I-elimination (1) holds. On the other hand, (H1) gives for $p : \mathrm{I}_A(a,a)$, that

$$\tau_{a,a,f(p)} \approx \mathrm{ref}(a). \tag{2}$$

With (1) this gives

$$p \approx \mathrm{ref}(a)$$

for any $p : \mathrm{I}_A(a,a)$, which is equivalent to UIP for A. ◀

▶ **Corollary 3.** *Assuming any E-category with A as the type of objects can be extended to an H-category. Then UIP holds for A.*

In classical category theory any category may be equipped with isomorphism as equality of objects (see remark above). This is thus *not* possible in basic type theory, with the above A^ι as counter examples.

5 Categories in Univalent Type Theory

In Univalent Type Theory [9], a *set* is a type that satisfies the UIP condition. A *precategory* [9, Chapter 9.1] is a tuple $\mathcal{C} = (C, \mathrm{Hom}, \circ, 1)$ where C is a type, Hom is a family of types over $C \times C$, such that $\mathrm{Hom}(a,b)$ is a set for all $a, b : C$. Moreover $1_a : \mathrm{Hom}(a,a)$ and

$$\circ : \mathrm{Hom}(b,c) \times \mathrm{Hom}(a,b) \to \mathrm{Hom}(a,c)$$

satisfy the associativity and unit laws up to I-equality.

Such a precategory thus forms an E-category by considering the hom-set as the setoid $(\mathrm{Hom}(a,b), \mathrm{I}_{\mathrm{Hom}(a,b)}(\cdot,\cdot))$. We have moreover:

▶ **Theorem 4.** *Every precategory whose type of objects is a set is an H-category.*

Proof. Define $a \cong b$ to be the statement that a and b are isomorphic in $\mathcal{C}$ i.e.

$$(\exists f : \mathrm{Hom}(a,b))(\exists g : \mathrm{Hom}(b,a))\, g \circ f \doteq 1_a \wedge f \circ g \doteq 1_b.$$

By I-elimination one defines a function

$$\sigma_{a,b} : a \doteq b \to a \cong b \tag{3}$$

by $\sigma_{a,a}(\mathrm{ref}(a)) = (1_a, (1_a, (\mathrm{ref}(1_a), \mathrm{ref}(1_a))))$. Define by taking the first projection $\tau_{a,b,p} = (\sigma_{a,b}(p))_1 : \mathrm{Hom}(a,b)$. By I-induction it follows that
$\tau_{a,a,\mathrm{ref}(a)} \doteq 1_a$ for any $p : a \doteq a$,
$\tau_{b,c,q} \circ \tau_{a,b,p} \doteq \tau_{a,c,q\circ p}$ for any $p : a \doteq b$ and $q : b \doteq c$.

For a precategory where C is a set, it follows that for any $p, q : a \doteq b$ such that $p \doteq q$ holds, so by substitution

$$\tau_{a,b,p} = \tau_{a,b,q}.$$

Thus τ gives an H-structure on C, so the precategory is in fact an H-category. ◀

An *univalent category,* or UF-category, is a precategory where the function $\sigma_{a,b}$ in (3) is an equivalence for any $a, b : C$; see [2] and [9, Chapter 9.1]. In particular, it means that if $a \cong b$, then $\mathrm{I}_C(a,b)$.

▶ **Example 5.** An example of a precategory which is not a univalent category is given by $C = \mathrm{N}_2$ where $\mathrm{Hom}(m,n) = \mathrm{N}_1$. Here $0 \cong 1$, but $\mathrm{I}_C(0,1)$ is false.

▶ Remark. Note that a UF-category whose type of objects is a set, is a skeletal H-category.

The reverse is however not true.

▶ **Example 6.** Suppose that $\mathcal{C}$ is a skeletal precategory whose type of objects is a set. Is $\mathcal{C}$ necessarily a univalent category? No. Consider the group $\mathbb{Z}_2$ as a one object, skeletal precategory: Let the underlying set be N_1 and $\mathrm{Hom}(0,0) = \mathrm{N}_2$ with 0 as unit and $\circ$ as addition. This is not a univalent category, compare Example 9.15 in [9]. Thus the standard multiplication table presentation of a nontrivial group is not a univalent category.

6 Conclusion

In conclusion, the notion of univalent category is too restrictive to cover many familiar examples. H-category is generalization of precategory and is a convenient version of E-category with equality on objects. The notion of E-category is still more general as shown here.

References

1 Peter Aczel. Galois: a theory development project. Report for the 1993 Turin meeting on the Representation of Mathematics in Logical Frameworks, 1995. URL: `http://www.cs.man.ac.uk/~petera/papers.html`.

2 Benedikt Ahrens, Krzysztof Kapulkin, and Michael Shulman. Univalent categories and the Rezk completion. *Math. Structures Comput. Sci.*, 25(5):1010–1039, 2015.

3 Gilles Barthe, Venanzio Capretta, and Olivier Pons. Setoids in type theory. *Journal of Functional Programming*, 13:261–293, 2003.

4 Errett Bishop. *Foundations of constructive analysis*. McGraw-Hill Book Co., New York-Toronto, Ont.-London, 1967.

5 Saunders Mac Lane. *Categories for the working mathematician*, volume 5 of *Graduate Texts in Mathematics*. Springer-Verlag, New York, second edition, 1998.

6 Erik Palmgren. Proof-relevance of families of setoids and identity in type theory. *Arch. Math. Logic*, 51(1-2):35–47, 2012.

7 Erik Palmgren. Constructions of categories of setoids from proof-irrelevant families. *Arch. Math. Logic*, 56(1-2):51–66, 2017.

8 Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. *Log. Methods Comput. Sci.*, 10(3):3:25, 14, 2014.

9 The Univalent Foundations Program. *Homotopy type theory—univalent foundations of mathematics*. The Univalent Foundations Program, Princeton, NJ; Institute for Advanced Study (IAS), Princeton, NJ, 2013.

10 Olov Wilander. Constructing a small category of setoids. *Math. Structures Comput. Sci.*, 22(1):103–121, 2012.