24th International Conference on Types for Proofs and Programs

TYPES 2018, June 18–21, 2018, Braga, Portugal

Edited by

Peter Dybjer

José Espírito Santo

Luís Pinto

Editors

Peter Dybjer
Department of Computer Science and Engineering
Chalmers University of Technology
Göteborg, Sweden
peterd@chalmers.se

José Espírito Santo
Centro de Matemática
Universidade do Minho
Braga, Portugal
jes@math.uminho.pt

Luís Pinto
Centro de Matemática
Universidade do Minho
Braga, Portugal
luis@math.uminho.pt

ACM Classification 2012
Theory of computation → Type theory; Theory of computation → Constructive mathematics; Theory of computation → Logic and verification; Theory of computation → Program verification; Software and its engineering → Functional languages

ISBN 978-3-95977-106-1

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at https://www.dagstuhl.de/dagpub/978-3-95977-106-1.

Publication date
November, 2019

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

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2018.0
ISBN 978-3-95977-106-1 **ISSN 1868-8969** **https://www.dagstuhl.de/lipics**

LIPIcs – Leibniz International Proceedings in Informatics

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

ISSN 1868-8969

https://www.dagstuhl.de/lipics

Contents

24th International Conference on Types for Proofs and Programs (TYPES 2018).
Editors: Peter Dybjer, José Espírito Santo, and Luís Pinto

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

Preface

This volume is the post-proceedings of the 24th International Conference on Types for Proofs and Programs, TYPES 2018, which was held at Universidade do Minho in Braga, Portugal, between the 18th and the 21st of June in 2018.

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 2018 meeting in Braga, 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), Novi Sad (2016), and Budapest (2017) with post-proceedings published in various outlets, with the last six 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 2018 included four invited talks by Cédric Fournet (Microsoft Research, UK) on Building Verified Cryptographic Components Using F*, Delia Kesner (IRIF CNRS and Université Paris-Diderot, France) on Multi Types for Higher-Order Languages, Matthieu Sozeau (INRIA, France) on The Predicative, Polymorphic Calculus of Cumulative Inductive Constructions and its Implementation, and Josef Urban (CIIRC, Czech Republic) on Machine Learning for Proof Automation and Formalization. The contributed part of the programme consisted of 42 talks. One of the sessions of the programme payed tribute to Martin Hofmann, and included three of the contributed talks, and an invited talk by Ralph Matthes (CNRS, IRIT, University of Toulouse, France). The conference was attended by more than 80 researchers.

TYPES 2018 was sponsored by the COST Action CA15123 EUTypes, supported by COST (European Cooperation in Science and Technology), Centro de Matemática da Universidade do Minho, Conselho Cultural da Universidade do Minho, and Câmara Municipal de Braga.
The call for contributions to the post-proceedings of TYPES 2018 was open and not restricted to the authors and presentations of the conference. Out of 8 submitted papers, 7 were selected after several rounds of refereeing; the final decisions were made by the editors. The papers span a wide range of interesting topics: Bishop's set theory; meta-theory of logics and type systems and its formalisation; models of cubical type theory; normalization by evaluation; non-strictly positive data types; program verification; semantic subtyping. We thank both the authors and the reviewers for their hard work.

Peter Dybjer, José Espírito Santo, and Luís Pinto
September 2019

24th International Conference on Types for Proofs and Programs (TYPES 2018).
Editors: Peter Dybjer, José Espírito Santo, and Luís Pinto
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

List of Authors

Davide Ancona
DIBRIS, Università di Genova, Italy

Ulrich Berger
Dept. of Computer Science, Swansea University, United Kingdom
u.berger@swansea.ac.uk

Giuseppe Castagna
CNRS, IRIF, Université Paris Diderot, France

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

Ralph Matthes
IRIT (CNRS and University of Toulouse), France
Ralph.Matthes@irit.fr

Iosif Petrakis
Ludwig-Maximilians-Universität Munich, Germany
petrakis@math.lmu.de

Tommaso Petrucciani
DIBRIS, Università di Genova, Italy and IRIF, Université Paris Diderot, France

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

Anders Schlichtkrull
DTU Compute - Department of Applied Mathematics and Computer Science, Technical University of Denmark, Denmark
andschl@dtu.dk

Filippo Sestini
Functional Programming Laboratory, University of Nottingham, United Kingdom
filippo.sestini@nottingham.ac.uk

Anton Setzer
Dept. of Computer Science, Swansea University, United Kingdom
a.g.setzer@swansea.ac.uk

Taichi Uemura
University of Amsterdam, The Netherlands
t.uemura@uva.nl

Elena Zucca
DIBRIS, Università di Genova, Italy

24th International Conference on Types for Proofs and Programs (TYPES 2018).
Editors: Peter Dybjer, José Espírito Santo, and Luís Pinto
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Martin Hofmann's Case for Non-Strictly Positive Data Types

Ulrich Berger
Dept. of Computer Science, Swansea University, United Kingdom
u.berger@swansea.ac.uk

Ralph Matthes
IRIT (CNRS and University of Toulouse), France
Ralph.Matthes@irit.fr

Anton Setzer
Dept. of Computer Science, Swansea University, United Kingdom
a.g.setzer@swansea.ac.uk

Abstract

We describe the breadth-first traversal algorithm by Martin Hofmann that uses a non-strictly positive data type and carry out a simple verification in an extensional setting. Termination is shown by implementing the algorithm in the strongly normalising extension of system F by Mendler-style recursion. We then analyze the same algorithm by alternative verifications first in an intensional setting using a non-strictly positive inductive definition (not just a non-strictly positive data type), and subsequently by two different algebraic reductions. The verification approaches are compared in terms of notions of simulation and should elucidate the somewhat mysterious algorithm and thus make a case for other uses of non-strictly positive data types. Except for the termination proof, which cannot be formalised in Coq, all proofs were formalised in Coq and some of the algorithms were implemented in Agda and Haskell.

2012 ACM Subject Classification Theory of computation → Logic and verification; Theory of computation → Type theory; Software and its engineering → Abstract data types; Software and its engineering → Recursion; Software and its engineering → Software verification; Software and its engineering → Coroutines

Keywords and phrases non strictly-positive data types, breadth-first traversal, program verification, Mendler-style recursion, System F, theorem proving, Coq, Agda, Haskell

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.1

Supplement Material `https://github.com/rmatthes/breadthfirstalahofmann`

Funding The second and third authors got financial support by the COST action CA15123 EUTYPES.
Ulrich Berger: Work supported by CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962, CID FP7 Marie Curie International Research Project, H2020-MSCA-RISE-2016-731143
Anton Setzer: Work supported by EPSRC grant EP/G033374/1 *Theory and Applications of Induction Recursion*; CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962, CID FP7 Marie Curie International Research Project, H2020-MSCA-RISE-2016-731143

Acknowledgements This paper is dedicated to the memory of the late Martin Hofmann. Martin was one of the leading researchers in the field of functional programming and type theory. This article is based on his notes [6, 7], which is only one example of the inspiration he has given to many researchers. His tragic unexpected death was a deep loss for the community.

1 Introduction

Given a finitely-branching tree t with labels at all nodes there are different ways to traverse it starting with its root. Depth-first traversal first goes along the entire left-most[1] branch until the leaf is reached and then backtracks and pursues with the next sibling. An efficient implementation of depth-first traversal is possible by using a stack of entry points into subtrees of t. In the beginning, t is pushed on the stack. While the stack is non-empty, a tree is popped from it, its root visited and its children pushed on the stack from right to left. If the tree is infinite, depth-first traversal does not visit all nodes in most cases. In particular, if the left-most branch is infinite, the algorithm will be confined to traverse this branch. (It visits all nodes if and only if all branches different from the right-most branch are finite.)

The described problem does not occur with breadth-first traversal. The latter means that it first visits the root, then the roots of all immediate subtrees from left to right[2], then in turn the roots of their immediate subtrees from left to right, etc. An efficient implementation is given by way of an efficiently implemented first-in, first-out queue (FIFO). The description of the algorithm is as before for depth-first traversal, but now with the FIFO operations. However, the immediate subtrees of the currently treated tree are put into the queue from left to right.

While these algorithms are easy to provide in imperative languages with worst-case linear execution time, functional programming languages only easily provide amortized linear execution time for the breadth-first traversal. (In functional programming, the "traversal" is replaced by the task to construct the list of all node labels in the order the imperative algorithm would traverse them.) Okasaki [12] presented for the first time an elegant and worst-case constant-time functional implementation of FIFO, thus yielding worst-case linear-time breadth-first traversal. However, there are also different functional implementations with worst-case linear time [8].

This paper is about breadth-first traversal in a functional programming language, but efficiency is not the concern here. Instead, we explore an algorithm for breadth-first traversal invented by Martin Hofmann, as presented in his posting [6] to the `TYPES forum` mailing list. In a draft [7], Martin Hofmann shows how he crafted the data type on which his proposal is based. There one also finds a sketch of a correctness proof by induction over binary trees.

We will first explain what is so special about Hofmann's algorithm. In dependent type theory one normally wants all programs to be terminating, i. e., the terms to be strongly normalizing. A well-established way of ensuring strong normalization is to restrict recursion to structural recursion on inductive structures obtained as least fixed points of monotone operators. Monotonicity is usually replaced by the stronger syntactic condition of positivity, which means that the expression that describes the operation must have its formal parameter at positive positions only. Positivity does not exclude going twice to the left of the arrow for the function type – only strict positivity would forbid that, but that latter is imposed in most implementations of type theory, including the Coq system and Agda. Non-strictly positive data types may not have a naive set-theoretic semantics [15], but they exist well in system F [3], i. e., polymorphic lambda-calculus [14], where they can be encoded as weakly initial algebras, in other words, as data types with constructors together with an iterator for programming structurally recursive functions. As evaluation in system F is strongly normalizing, all those structurally recursive programs are terminating.

[1] The choice of the left direction is only for definiteness of our description.
[2] This is again just for definiteness.

Hofmann's algorithm is based on the following non-strictly positive data type (our notation):

$$\begin{array}{l}\mathsf{Inductive}\ \mathsf{Rou} :=\\ \quad|\ \ \mathsf{Over}\ \ :\ \ \mathsf{Rou}\\ \quad|\ \ \mathsf{Next}\ \ :\ \ ((\mathsf{Rou}\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N})\to\mathsf{Rou}\end{array}$$

Rou stands for "routine", and there is the constructor Over for the routine that is not executing further, and the crucial non-strictly positive constructor Next that takes a functional of type $(\mathsf{Rou}\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}$ as argument to yield a composite routine.[3] Rou appears at a position twice to the left of $\to$ in the type of the argument, hence positively. As we mentioned above, such inductive definitions are ruled out in most proof assistants, notably in the Coq system and in Agda.

While there is a generic iterator for Rou in system F – as mentioned before – the recursive functions needed for the algorithm are not all instances of the iterator. Functions that would calculate the same values can be defined by iteration, but they would not reflect the algorithm properly. However, this shortcoming can be solved by using recursive functions in the style of Mendler [11] which can be provided by a (mild) extension of system F. A detailed account of these issues, which also settles the question of termination, is given in Sect. 5. Besides that, the paper concentrates on different correctness proofs, most of them based on simulations by related algorithms using different intermediate data types, with the aim to reveal and explain the internal structure of Hofmann's algorithm and to replace the impredicative type Rou by a predicative type while preserving the structural characteristics of the original algorithm.

Overview of the paper: After presenting an executable specification of breadth-first traversal as the concatenation of all levels (niveaux) of a tree (Sect. 2) we introduce the data type of routines and Hofmann's algorithm breadthfirst (Sect. 3) and prove its partial correctness (i. e., correctness assuming termination) following Hofmann's proof sketch (Sect. 4). Termination is proven in Sect. 5 by implementing the functions and data types in the strongly normalising extension of system F by Mendler-style recursion.

Having thus set the stage, we dive into the analysis of Hofmann's algorithm. We begin with a correctness proof (Sect. 6) based on a non-strictly positive inductive representation relation between routines and double lists (lists of lists) that does not require auxiliary functions. This proof does not require extensionality which is a natural prerequisite for Hofmann's correctness proof. Next we present a proof based on the natural extension of breadth-first traversal to forests (lists of trees) providing interesting insight into the internal structure of Hofmann's algorithm (Sect. 7). We give a meaning to the routine corresponding to a forest ts. It is the routine $(\mathsf{c}\,ts)$ computing the traversal of a forest ts while recursively calling $(\mathsf{c}\,(\mathsf{sub}\,ts))$ for the immediate subforest $(\mathsf{sub}\,ts)$ of ts. The function extract evaluates these recursive functions, and the function br in Hofmann's algorithm, that initially seems to be mysterious, is decoded as an operation which computes $(\mathsf{c}\,(t :: ts))$ from $(\mathsf{c}\,ts)$ and t.

Building on this insight we construct two predicative versions of this algorithm. The first one introduced in Sect. 8 is based on the observation that the routines occurring in the algorithm can be represented as lists of functions $\mathsf{List}\,\mathbb{N}\to\mathsf{List}\,\mathbb{N}$. Therefore we can replace the impredicative data type Rou by the predicative type $\mathsf{Rou}' := \mathsf{List}\,(\mathsf{List}\,\mathbb{N}\to\mathsf{List}\,\mathbb{N})$.

[3] $\mathsf{List}\,\mathbb{N}$ is the type of lists of natural numbers which are taken here for simplicity; any list type would be fine. The data type is tailor-made to our breadth-first traversal problem that requires to compute an element of $\mathsf{List}\,\mathbb{N}$.

Meaning is given to the routine corresponding to a forest ts as the routine $\mathsf{traverse}\, ts : \mathsf{Rou}'$ which is the list of functions appending the levels of the forest. As before, the function br' corresponding to br computes $(\mathsf{traverse}\,(t :: ts))$ from $(\mathsf{traverse}\, ts)$. The second predicative version (Sect. 9) observes that the functions in Rou' constructed in the algorithms are append functions, i. e., functions of the form $\lambda l\,.\,l' +\!\!+ l$. They can be represented as lists of natural numbers, so we can replace Rou' by the simpler type $\mathsf{Rou}'' := \mathsf{List}^2\,\mathbb{N}$ of double lists. These double lists correspond to the list of levels in the specification of breadth-first traversal.

The findings are summarized in Sect. 10 where we show that the various algorithms and proofs all have the structure of a "simulation of systems". In addition we show that the two predicative algorithms provide a splitting of Hofmann's algorithm into two simpler phases. We round the paper off with a discussion of and pointers to the implementation and formalization of our work in the proof assistants Coq and Agda, highlighting the difficulties caused by non-strict positivity and how to overcome them (Sect. 11), and conclude with a reflection on what was achieved and an outlook to a possible extension of the domain of the algorithms to infinite trees.

2 Specification of breadth-first traversal

We fix the simplest setting to express the task of programming breadth-first traversal: our trees are not arbitrarily finitely branching but just binary, and they are even finite. As did Hofmann, we put labels on the inner nodes and the leaves. For simplicity, we restrict the type of labels to be the natural numbers but any other type could be used instead.

$$
\begin{array}{l}
\text{Inductive } \mathsf{Tree} := \\
\quad | \quad \mathsf{Leaf} \quad : \quad \mathbb{N} \to \mathsf{Tree} \\
\quad | \quad \mathsf{Node} \quad : \quad \mathsf{Tree} \to \mathbb{N} \to \mathsf{Tree} \to \mathsf{Tree}
\end{array}
$$

We use the typing conventions

n	:	$\mathbb{N}$
l	:	$\mathsf{List}\,\mathbb{N}$
ls	:	$\mathsf{List}^2\mathbb{N} \stackrel{\text{Def}}{=} \mathsf{List}\,(\mathsf{List}\,\mathbb{N})$
t, tl, tr	:	Tree (tl and tr are typically used for the left and right subtree, respectively)

An extended use is made of the auxiliary function zip that "zips" the successive lists in both arguments using the append function for lists (denoted by $+\!\!+$). More precisely, our zip behaves like `zipWith (++)` (with `zipWith` in the Haskell basic library, and `(++)` the Haskell notation for append viewed as a function) for arguments of equal lengths but if lengths differ zip extends the shorter argument with empty lists wheras `zipWith (++)` truncates the longer argument.

$$
\begin{array}{l}
\mathsf{zip} : \mathsf{List}^2\mathbb{N} \to \mathsf{List}^2\mathbb{N} \to \mathsf{List}^2\mathbb{N} \\
\mathsf{zip}\,[]\,ls = ls \qquad \mathsf{zip}\,(l :: ls)\,[] = l :: ls \qquad \mathsf{zip}\,(l :: ls)\,(l' :: ls') = (l +\!\!+ l') :: \mathsf{zip}\, ls\, ls'
\end{array}
$$

▶ **Lemma 1** (basic properties of zip).

(a) $\mathsf{zip}\, ls\, [] = ls$.

(b) $\mathsf{zip}\, ls_1\, (\mathsf{zip}\, ls_2\, ls_3) = \mathsf{zip}\, (\mathsf{zip}\, ls_1\, ls_2)\, ls_3$.

We create the list of labels for every horizontal section of the tree, starting with its root (niv refers to the French word "niveaux" for levels – the function collects the labels level-wise).

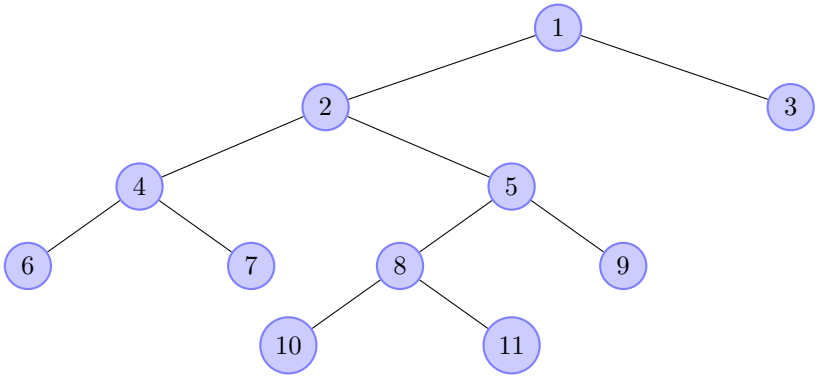
$$\mathsf{niv} : \mathsf{Tree} \to \mathsf{List}^2\mathbb{N}$$
$$\mathsf{niv}\,(\mathsf{Leaf}\,n) = [[n]] \qquad \mathsf{niv}\,(\mathsf{Node}\,t_1\,n\,t_2) = [n] :: \mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2)$$

From the definition, we see that niv is compositional, which the breadth-first traversal function is not (as also remarked in Hofmann's draft [7]). The latter is defined as follows:

$$\mathsf{breadthfirst}_{\mathsf{spec}} : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{breadthfirst}_{\mathsf{spec}}\,t = \mathsf{flatten}\,(\mathsf{niv}\,t)$$

Here, $\mathsf{flatten} : \mathsf{List}^2\mathbb{N} \to \mathsf{List}\,\mathbb{N}$ denotes concatenation of all those lists (the monad multiplication of the list monad). We do not consider this description of $\mathsf{breadthfirst}_{\mathsf{spec}}$ as an implementation but as an executable specification.

▶ **Example 2.** Let t correspond to the following graphical representation:



Then $\mathsf{niv}\,t = [[1],[2,3],[4,5],[6,7,8,9],[10,11]]$ and $\mathsf{breadthfirst}\,t = [1,\ldots,11]$.

3 Definition of breadth-first traversal via routines

We again show the type Martin Hofmann came up with in his 1993 posting [6]:

$$\begin{array}{l}\mathsf{Inductive\ Rou} :=\\ \quad|\ \ \mathsf{Over} \ : \ \mathsf{Rou}\\ \quad|\ \ \mathsf{Next} \ : \ ((\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}\end{array}$$

The names of the constructors are not those chosen by Hofmann but were suggested to us by Olivier Danvy (since they are used for programming with coroutines). A routine of the form $(\mathsf{Next}\,f)$ comes with a functional f of type $(\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$ whose argument can be seen as a "continuation", and $f\,k$, with k such a continuation, denotes a list that could be the result of our breadth-first traversal problem. In general, elements of Rou should be seen as encapsulations of routines for the computation of lists of natural numbers.

We use the typing conventions

$$\begin{array}{lcl} c & : & \mathsf{Rou} \quad \text{(routines)}\\ k & : & \mathsf{Rou} \to \mathsf{List}\,\mathbb{N} \quad \text{(continuations)}\\ f & : & (\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}\end{array}$$

We define the following function (called apply by Hofmann) naively by pattern matching on its first argument and show that this is a legal definition of a terminating function below in Section 5:

$$\mathsf{unfold} : \mathsf{Rou} \to (\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{unfold}\,\mathsf{Over} = \lambda k\,.\,k\,\mathsf{Over} \qquad \mathsf{unfold}\,(\mathsf{Next}\,f) = f$$

The name unfold seems justified (and more intuitive than Hofmann's choice of name) for the second case of the definition since it unfolds (Next f) to its argument f. Unfolding Over is curious since it yields again an expression involving Over.

The traversal algorithm is expressed as a transformation on routines, instructed by the tree argument. It is by plain iteration on that tree argument ($\circ$ denotes composition of functions).

$$\begin{aligned}&\mathsf{br} : \mathsf{Tree} \to \mathsf{Rou} \to \mathsf{Rou}\\ &\mathsf{br}\,(\mathsf{Leaf}\,n)\,c &&= \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,k)\\ &\mathsf{br}\,(\mathsf{Node}\,tl\,n\,tr)\,c &&= \mathsf{Next}\,\big(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)\big)\end{aligned}$$

We define a function extract which computes a result from a given routine. Again, we naively define this function by pattern matching on the inductive type of routines, but we here allow ourselves a recursive call, as follows:

$$\mathsf{extract} : \mathsf{Rou} \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{extract}\,\mathsf{Over} = [] \qquad \mathsf{extract}\,(\mathsf{Next}\,f) = f\,\mathsf{extract}$$

What is noteworthy here is that the recursive call is not to extract with some term smaller than (Next f) in any sense. The term extract is even fed in as an argument to the term f, which is type-correct since extract is of the type of a continuation. In Section 5, we will show that this is a plain form of iteration, thus ensuring termination and well-definedness. As we are doing for unfold, we currently view the equations for extract as a specification, which allows us to carry out verification in the next section.

Hofmann's algorithm calculates the routine transformer br for the given tree, applies it to the trivial routine and then extracts the result from the output routine:

$$\mathsf{breadthfirst} : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$$
$$\mathsf{breadthfirst}\,t = \mathsf{extract}(\mathsf{br}\,t\,\mathsf{Over})$$

Of course, we have to make sure that breadthfirst is a total function and that its results agree with those of $\mathsf{breadthfirst}_{\mathsf{spec}}$.

4 Martin Hofmann's verification of partial correctness

Here, we follow the sketch in Hofmann's notes [6] and argue how functional correctness (i. e., the algorithm's result meets the specification) follows from the equational specification of unfold and extract and the definitions of the other functions (br and those used for the executable specification in Section 2).

We define a routine transformer that is instructed by a double list, by plain iteration on that list.

$$\gamma : \mathsf{List}^2\mathbb{N} \to \mathsf{Rou} \to \mathsf{Rou}$$
$$\gamma\,[]\,c = c \qquad \gamma\,(l :: ls)\,c = \mathsf{Next}\Big(\lambda k\,.\,l +\!\!+ \big(\mathsf{unfold}\,c\,(k \circ \gamma\,ls)\big)\Big)$$

The following three lemmas (stated in Hofmann's notes [6] without their simple proofs shown below) on the function γ are all the preparations needed for the proof of functional correctness (cf. Theorem 6).

▶ **Lemma 3.** $\mathsf{extract}\,(\gamma\,ls\,\mathsf{Over}) = \mathsf{flatten}\,ls$.

Proof. Induction on ls.
$\mathsf{extract}\,(\gamma\,[]\,\mathsf{Over}) = \mathsf{extract}\,\mathsf{Over} = [] = \mathsf{flatten}\,[]$.
$\mathsf{extract}\,(\gamma\,(l :: ls)\,\mathsf{Over}) = \mathsf{extract}\,(\mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (\mathsf{unfold}\,\mathsf{Over}\,(k \circ \gamma\,ls))))$
$= l +\!\!+ ((\mathsf{extract} \circ \gamma\,ls)\,\mathsf{Over}) \overset{\text{IH}}{=} l +\!\!+ \mathsf{flatten}\,ls = \mathsf{flatten}(l :: ls)$. ◀

By $\overset{ext}{=}$ we denote extensional, i. e., pointwise, equality of functions. The following lemma uses two instances of the principle of extensionality. The first states that functions $f : (\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}$ respect extensional equality, i. e., $k \overset{ext}{=} k'$ implies $f\,k = f\,k'$. The second states extensionality of the constructor $\mathsf{Next} : ((\mathsf{Rou} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}$ (w. r. t. extensional equality of its argument). The following two lemmas (4 and 5) and consequently Theorem 6 depend on extensionality for their proofs.

▶ **Lemma 4.** $\gamma\,ls \circ \gamma\,ls' \overset{ext}{=} \gamma\,(\mathsf{zip}\,ls\,ls')$.

Proof. Induction on ls and ls'.
$\gamma\,[] \circ \gamma\,ls' \overset{ext}{=} \gamma\,ls' = \gamma\,(\mathsf{zip}\,[]\,ls')$.
$\gamma\,ls \circ \gamma\,[] \overset{ext}{=} \gamma\,ls = \gamma\,(\mathsf{zip}\,ls\,[])$.
$\gamma\,(l :: ls)\,(\gamma\,(l' :: ls')\,c)$
$= \gamma\,(l :: ls)\,(\mathsf{Next}\,(\lambda k'\,.\,l' +\!\!+ (\mathsf{unfold}\,c\,(k' \circ \gamma\,ls'))))$
$= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (\mathsf{unfold}\,(\mathsf{Next}\,(\lambda k'\,.\,l' +\!\!+ (\mathsf{unfold}\,c\,(k' \circ \gamma\,ls'))))\,(k \circ \gamma\,ls))$
$= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (l' +\!\!+ (\mathsf{unfold}\,c\,(k \circ \gamma\,ls \circ \gamma\,ls'))))$
$= \mathsf{Next}\,(\lambda k\,.\,l +\!\!+ (l' +\!\!+ (\mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{zip}\,ls\,ls')))))$ (by ind. hyp. and extensionality)
$= \gamma\,((l +\!\!+ l') :: \mathsf{zip}\,ls\,ls')\,c$ (by associativity of $+\!\!+$)
$= \gamma\,(\mathsf{zip}\,(l :: ls)\,(l' :: ls'))\,c$. ◀

▶ **Lemma 5.** $\mathsf{br}\,t \overset{ext}{=} \gamma\,(\mathsf{niv}\,t)$.

Proof. Induction on t.
$\mathsf{br}\,(\mathsf{Leaf}\,n)\,c = \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,k) = \mathsf{Next}\,(\lambda k\,.\,[n] +\!\!+ (\mathsf{unfold}\,c\,k)) = \gamma\,[[n]]\,c$
$= \gamma\,(\mathsf{niv}\,(\mathsf{Leaf}\,n))\,c$.
$\mathsf{br}\,(\mathsf{Node}\,t_1\,n\,t_2)\,c = \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \mathsf{br}\,t_1 \circ \mathsf{br}\;t_2))$
$\overset{\text{IH, extensionality}}{=} \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{niv}\,t_1) \circ \gamma\,(\mathsf{niv}\,t_2)))$
$\overset{\text{Lem. 4, extensionality}}{=} \mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \gamma\,(\mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2))))$
$= \gamma\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,t_1)\,(\mathsf{niv}\,t_2))\,c = \gamma\,(\mathsf{niv}\,(\mathsf{Node}\,t_1\,n\,t_2))\,c$. ◀

From these lemmas, we now directly (without further inductive arguments) obtain the main result of this section.

▶ **Theorem 6.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$, *i. e., for all trees* t, *we have* $\mathsf{breadthfirst}\,t = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$.

Proof. $\mathsf{breadthfirst}\,t = \mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) \overset{\text{Lem. 5}}{=} \mathsf{extract}\,(\gamma\,(\mathsf{niv}\,t)\,\mathsf{Over})$
$\overset{\text{Lem. 3}}{=} \mathsf{flatten}\,(\mathsf{niv}\,t) = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$. ◀

This completes the proof based on the sketch by Martin Hofmann.

5 Termination of Hofmann's algorithm

In his 1993 posting [6] Martin Hofmann argued about the existence of the functions unfold and extract through an impredicative encoding of data types in system F, equipped with parametric equality (equality that is defined as a logical relation by induction over the type of terms being equated, which is impredicative for the case of the universal quantifier). This is, in our opinion, not fully satisfactory, since a verification with parametric equality only shows the existence of a function that yields breadth-first traversal but does not verify the termination of the algorithm itself that is expressed by the defining equations.

Like Martin Hofmann, we are heading for a language-based termination guarantee: We implement the data types and functions of this algorithm in system F extended by Mendler-style recursion, which is known to be strongly normalising. In fact, all relevant data types (including Rou) and all functions defined by iteration can be defined in plain system F in the usual way [4]. Mendler's extension is only needed to properly model the algorithmic behaviour of the function unfold.

We begin with the system F encodings of the type Rou and the function extract as an example of a plain iteration, since in these cases the encoding is very similar to Mendler's encoding.

If we strip off the names of the constructors so as to fit into the scheme of categorical data types[4], we get Rou as least fixed point of the "functor" RouF, defined on types by

$$\mathsf{RouF}\, A := 1 + ((A \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \ ,$$

with the one-element type 1 (a. k. a. unit type with only inhabitant $*$) and the type constructor $+$ for disjoint sums (with injections inl and inr and case analysis operator $[s_0, s_1] : A_0 + A_1 \to C$ for $s_i : A_i \to C$, $i = 0, 1$). Clearly, the type A only occurs at a non-strictly positive position in the right-hand side. The usual impredicative encoding of least fixed points in system F (also called "Church encoding") yields as least fixed point of RouF

$$\mathsf{Rou}_{\mathsf{Imp}} := \forall A\,.\,(\mathsf{RouF}\, A \to A) \to A \ .$$

Iteration over Rou is then given by "catamorphisms" for RouF-algebras since Rou itself is the carrier of the initial RouF-algebra. Beware that initiality holds only with respect to a categorical semantics. Computationally, one only gets weak initiality, that is, the existence but not the uniqueness of the morphism (given by the iterator) in the standard commuting diagram for initial algebras. Moreover, the single[5] equation expressed by the commuting diagram is computationally directed: we will later use the symbol $\rhd^*$ for that relation, instead of the symmetric $=$ that appears in traditional categorical modeling.

This weak initiality principle already captures the behaviour of extract (but we will have to define extract differently later since also unfold needs to be taken care of). The details are as follows: We define the iterator

$$\mathsf{RouIt} : \forall A\,.\,(\mathsf{RouF}\, A \to A) \to \mathsf{Rou}_{\mathsf{Imp}} \to A \qquad \mathsf{RouIt}\, A\, s\, t = t\, A\, s$$

[4] In the Haskell programming language, we would keep the constructors and define `data RouF a = Over | Next ((a -> List Nat) -> List Nat)`.

[5] before we make informal use of pattern matching that splits the rule into two rules

Due to positivity of RouF, there is a closed term $\mathsf{RouFmap}$, defined by case analysis on the sum as follows (slightly informally, for readability):

$$\begin{array}{lcl}
\mathsf{RouFmap} : \forall A, B\,.\,(A \to B) \to \mathsf{RouF}\,A \to \mathsf{RouF}\,B & & \\
\mathsf{RouFmap}\,A\,B\,h^{A\to B}\,(\mathsf{inl}\,u^{1}) & = & \mathsf{inl}\,u \\
\mathsf{RouFmap}\,A\,B\,h^{A\to B}\,(\mathsf{inr}\,f^{(A\to \mathsf{List}\,\mathbb{N})\to \mathsf{List}\,\mathbb{N}}) & = & \mathsf{inr}\,\big(\lambda k^{B\to \mathsf{List}\,\mathbb{N}}\,.\,f\,(k\circ h)\big)
\end{array}$$

This allows us to define the RouF-algebra $\mathsf{foldRou}_{\mathsf{Imp}}$ with carrier $\mathsf{Rou}_{\mathsf{Imp}}$:

$$\begin{array}{l}
\mathsf{foldRou}_{\mathsf{Imp}} : \mathsf{RouF}\,\mathsf{Rou}_{\mathsf{Imp}} \to \mathsf{Rou}_{\mathsf{Imp}} \\
\mathsf{foldRou}_{\mathsf{Imp}}\,t\,A\,s = s\,\big(\mathsf{RouFmap}\,\mathsf{Rou}_{\mathsf{Imp}}\,A\,(\mathsf{RouIt}\,A\,s)\,t\big)\ .
\end{array}$$

The impredicative implementations of the constructors, $\mathsf{Over}_{\mathsf{Imp}}$ and $\mathsf{Next}_{\mathsf{Imp}}$, are now instances of $\mathsf{foldRou}_{\mathsf{Imp}}$:

$$\begin{array}{lcll}
\mathsf{Over}_{\mathsf{Imp}} & := & \mathsf{foldRou}_{\mathsf{Imp}}\,(\mathsf{inl}\,*) & : \quad \mathsf{Rou}_{\mathsf{Imp}} \\
\mathsf{Next}_{\mathsf{Imp}} & := & \mathsf{foldRou}_{\mathsf{Imp}} \circ \mathsf{inr} & : \quad ((\mathsf{Rou}_{\mathsf{Imp}} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou}_{\mathsf{Imp}}
\end{array}$$

For convenience, we define ($\lambda\_$ is a void abstraction over unit type):

$$\begin{array}{l}
\mathsf{RouIt}_{\mathsf{Imp}} : \forall A\,.\,A \to (((A \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N}) \to A) \to \mathsf{Rou}_{\mathsf{Imp}} \to A \\
\mathsf{RouIt}_{\mathsf{Imp}}\,A\,s_0\,s_1 = \mathsf{RouIt}\,A\,[\lambda_\,.\,s_0\,,\ s_1]
\end{array}$$

We will write $\rhd$ for the one-step reduction relation of system F and $\rhd^*$ for its reflexive transitive closure. The characteristic reduction behaviour of $\mathsf{RouIt}_{\mathsf{Imp}}$ is given by

$$\begin{array}{lcl}
\mathsf{RouIt}_{\mathsf{Imp}}\,A\,s_0\,s_1\,\mathsf{Over}_{\mathsf{Imp}} & \rhd^* & s_0 \\
\mathsf{RouIt}_{\mathsf{Imp}}\,A\,s_0\,s_1\,(\mathsf{Next}_{\mathsf{Imp}}\,f) & \rhd^* & s_1\Big(\lambda k^{A\to \mathsf{List}\,\mathbb{N}}.\,f\big(k\circ(\mathsf{RouIt}_{\mathsf{Imp}}\,A\,s_0\,s_1)\big)\Big)
\end{array}$$

We can implement $\mathsf{extract}$, using the iterator with $A := \mathsf{List}\,\mathbb{N}$:

$$\begin{array}{l}
\mathsf{extract}_{\mathsf{Imp}} : \mathsf{Rou}_{\mathsf{Imp}} \to \mathsf{List}\,\mathbb{N} \\
\mathsf{extract}_{\mathsf{Imp}} = \mathsf{RouIt}_{\mathsf{Imp}}\,(\mathsf{List}\,\mathbb{N})\,[]\,\big(\lambda g^{(\mathsf{List}\,\mathbb{N}\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}\,.\,g(\lambda l\,.\,l)\big)
\end{array}$$

and obtain proper recursive behaviour with three subsequent steps of β-reduction and one η-reduction step (that can be assumed in Church-style versions of system F):

$$\mathsf{extract}_{\mathsf{Imp}}\,\mathsf{Over}_{\mathsf{Imp}} \rhd^* []\qquad \mathsf{extract}_{\mathsf{Imp}}\,(\mathsf{Next}_{\mathsf{Imp}}\,f) \rhd^* f\,\mathsf{extract}_{\mathsf{Imp}}$$

The equational specification of unfold may seem innocuous, but Harper and Mitchell [5] have shown that even rewrite rules that just have the form of a projection may break termination when added to system F. Consider the type $S := \forall A, B\,.\,(A \to A) \to B \to B$, which is trivially inhabited by a term that maps constantly to the identity on B. A different inhabitant J' of S is *added* to system F, and the reduction relation of system F is extended by a specific rule for J': $J'\,A\,A\,f^{A\to A} \rhd f$ for any type A. It is easy to construct a term in this extension that rewrites in several steps to itself, hence creating an infinite loop.[6] However, unfold *is* terminating, albeit not for trivial reasons.

We use the extension of system F by Mendler-style recursion which is strongly normalizing [11]. Already Mendler's original work accommodates non-strictly positive inductive types, as our Rou, but it was later shown that even that restriction to positivity is not necessary for

[6] This is also presented in detail in a paper by the second author [10, p.122], together with a discussion of a variant of the scheme of inductive types with iteration for which termination fails.

strong normalization (see [9, Section 6.1.1] for a semantic and [1] for a syntactic proof). We describe only the instance of Mendler-style primitive recursion that governs the data type $\mathsf{Rou}_{\mathsf{Men}}$, which is the one obtained for RouF. Mendler's extension permits the construction of a RouF-algebra $\mathsf{foldRou}_{\mathsf{Men}}$ with carrier $\mathsf{Rou}_{\mathsf{Men}} \stackrel{\text{Def}}{=} \mu\,\mathsf{RouF}$ (with μ in the sense of Mendler), i. e., we have

$$\mathsf{foldRou}_{\mathsf{Men}} : \mathsf{RouF}\,\mathsf{Rou}_{\mathsf{Men}} \to \mathsf{Rou}_{\mathsf{Men}} \text{ with recursor } \mathsf{RouRec} : \forall A\,.\,\mathsf{Step}_{\mathsf{Men}}\,A \to \mathsf{Rou}_{\mathsf{Men}} \to A$$

where the type of step functions is

$$\mathsf{Step}_{\mathsf{Men}} A := \forall X\,.\,(X \to \mathsf{Rou}_{\mathsf{Men}}) \to (X \to A) \to \mathsf{RouF}\,X \to A \ .$$

A step function $s : \mathsf{Step}_{\mathsf{Men}}\,A$ transforms a function $X \to A$ into a function $\mathsf{RouF}\,X \to A$, possibly using a function $X \to \mathsf{Rou}_{\mathsf{Men}}$. RouRec takes a step function and then transforms elements of $\mathsf{Rou}_{\mathsf{Men}}$ into elements of A. We have the rewrite rule

$$\mathsf{RouRec}\,A\,s\,(\mathsf{foldRou}_{\mathsf{Men}}\,t) \rhd s\,\mathsf{Rou}_{\mathsf{Men}}\,(\lambda x^{\mathsf{Rou}_{\mathsf{Men}}}\,.\,x)\,(\mathsf{RouRec}\,A\,s)\,t \ .$$

The individual constructors for $\mathsf{Rou}_{\mathsf{Men}}$ are obtained as in the impredicative encoding: $\mathsf{Over}_{\mathsf{Men}} := \mathsf{foldRou}_{\mathsf{Men}}\,(\mathsf{inl}\,*)$ and $\mathsf{Next}_{\mathsf{Men}}\,f := \mathsf{foldRou}_{\mathsf{Men}}\,(\mathsf{inr}\,f)$. Define the step terms for $\mathsf{extract}$ and unfold as follows (which could be mapped to terms of system F with unit and sum types):

$$\begin{array}{lll}
\mathsf{s}_{\mathsf{extract}} : \mathsf{Step}_{\mathsf{Men}}\,(\mathsf{List}\,\mathbb{N}) & & \\
\mathsf{s}_{\mathsf{extract}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to\mathsf{List}\,\mathbb{N}}\,(\mathsf{inl}\,u^{1}) & = & [] \\
\mathsf{s}_{\mathsf{extract}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to\mathsf{List}\,\mathbb{N}}\,(\mathsf{inr}\,f^{(X\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}) & = & f\,r \\
\mathsf{s}_{\mathsf{unfold}} : \mathsf{Step}_{\mathsf{Men}}\,\mathsf{A}_{\mathsf{unfold}} \quad \text{where } \mathsf{A}_{\mathsf{unfold}} := (\mathsf{Rou}_{\mathsf{Men}} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{List}\,\mathbb{N} & & \\
\mathsf{s}_{\mathsf{unfold}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to A_{\mathsf{unfold}}}\,(\mathsf{inl}\,u^{1}) & = & \lambda k\,.\,k\,\mathsf{Over}_{\mathsf{Men}} \\
\mathsf{s}_{\mathsf{unfold}}\,X\,i^{X\to\mathsf{Rou}_{\mathsf{Men}}}\,r^{X\to A_{\mathsf{unfold}}}\,(\mathsf{inr}\,f^{(X\to\mathsf{List}\,\mathbb{N})\to\mathsf{List}\,\mathbb{N}}) & = & \lambda k\,.\,f\,(k\circ i)
\end{array}$$

Define the Mendler-style implementations:

$$\begin{array}{lcllcl}
\mathsf{extract}_{\mathsf{Men}} & : & \mathsf{Rou}_{\mathsf{Men}} \to \mathsf{List}\,\mathbb{N} & \mathsf{extract}_{\mathsf{Men}} & = & \mathsf{RouRec}\,(\mathsf{List}\,\mathbb{N})\,\mathsf{s}_{\mathsf{extract}} \\
\mathsf{unfold}_{\mathsf{Men}} & : & \mathsf{Rou}_{\mathsf{Men}} \to \mathsf{A}_{\mathsf{unfold}} & \mathsf{unfold}_{\mathsf{Men}} & = & \mathsf{RouRec}\,\mathsf{A}_{\mathsf{unfold}}\,\mathsf{s}_{\mathsf{unfold}}
\end{array}$$

Obviously, $\mathsf{extract}_{\mathsf{Men}}\,\mathsf{Over}_{\mathsf{Men}} \rhd^* []$, $\mathsf{extract}_{\mathsf{Men}}\,(\mathsf{Next}_{\mathsf{Men}}\,f) \rhd^* f\,\mathsf{extract}_{\mathsf{Men}}$ (as for the impredicative implementation) and $\mathsf{unfold}_{\mathsf{Men}}\,\mathsf{Over}_{\mathsf{Men}} \rhd^* \lambda k\,.\,k\,\mathsf{Over}_{\mathsf{Men}}$. Finally,

$$\mathsf{unfold}_{\mathsf{Men}}\,(\mathsf{Next}_{\mathsf{Men}}\,f) \rhd^* \lambda k\,.\,f\,(k\circ(\lambda x\,.\,x)) \rhd^* f \ ,$$

where the latter reduction has one β- and two η-reduction steps at the end. Thus, $\mathsf{extract}_{\mathsf{Men}}$ and $\mathsf{unfold}_{\mathsf{Men}}$ are implementations of Hofmann's functions, and the original defining equations become reductions in the sense of $\rhd^*$ of the Mendler-style extension of system F.

Of course, one can also encode any algebraic data types such as lists and trees and functions defined by iteration on elements of such types in Mendler's system. This can be done in a similar (but simpler) way as sketched above for Rou and $\mathsf{extract}$ in plain system F. Moreover, the interpretation is algorithmically faithful to the equational specification of these functions in the sense that the defining equations become one or more term rewriting steps in Mendler's terminating system. In summary we have the following

▶ **Theorem 7.** *The data types and functions involved in Hofmann's algorithm for breadth-first traversal can be algorithmically faithfully interpreted in the strongly normalising system of Mendler-style recursion. Therefore, Hofmann's algorithm is terminating.*

6 Verification by a non-strictly positive inductive relation

We now embark on giving alternative correctness proofs of Hofmann's algorithm. They explore different concepts and provide different intuitions for the correctness of this algorithm (see Section 10 for a mathematical assessment of their relations). The first and mathematically most challenging alternative proof given in this section uses a non-strictly positive inductive relation between routines $c : \mathsf{Rou}$ and double lists $ls : \mathsf{List}^2\mathbb{N}$ that, intuitively, states that c "represents" ls.

First, we define when a continuation k is an *extractor* for a binary relation $R \subseteq \mathsf{Rou}\times\mathsf{List}^2\mathbb{N}$ (seen as a candidate for a representation relation) and an "initial" double list ls'.

$$\mathsf{isextractor}(R, ls', k) \overset{\text{Def}}{\equiv} \forall c, ls''\,.\, R(c, ls'') \to k\, c = \mathsf{flatten}\,(\mathsf{zip}\, ls'\, ls'')\ .$$

The fact that R occurs negatively in the formula $\mathsf{isextractor}(R, ls', k)$ means that the weaker R is the more constraints are imposed in order for k to be an extractor for R and ls'. The name "extractor" should convey the intuition that continuation k "extracts" the "right" result for ls'' out of routines c representing ls'' in the sense of R with initialization ls'. Note that the formula for the prescribed result does not mention the niv operation of the original specification $\mathsf{breadthfirst}_{\mathsf{spec}}$. Lemma 8 below shows that $\mathsf{extract}$ is an extractor for a suitable representation relation R and initialization $ls' = []$.

With this auxiliary concept of extractor (which, after all, is only an abbreviation for a rather short formula of logic) we now define the representation relation $\mathsf{rep} \subseteq \mathsf{Rou} \times \mathsf{List}^2\mathbb{N}$ inductively by two rules. Not surprisingly, rep takes the role of relation R in the foregoing definition. The reason why we formulated the notion of an extractor with a general relation argument R is that this allows us to conveniently express the induction principle for rep (as can be seen in the proof of Lemma 8 below). The inductive definition of rep is as follows:

$$\frac{}{\mathsf{rep}(\mathsf{Over}, [])}\ (\text{over})$$

$$\frac{\forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\, k = l +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\, ls'\, ls)}{\mathsf{rep}(\mathsf{Next}\, f, l :: ls)}\ (\text{next})$$

where in (next) the variables f, l, ls are implicitly universally quantified. The premise of the rule (next) contains the predicate rep positively (though not strictly positively) and therefore depends monotonically on it. By Tarski's fixed point theorem it follows that the smallest relation rep closed under the rules (over) and (next) exists.

Note that, since the premise of the rule (next) refers only to the result of applying f to k, the predicate rep respects extensional equality in the sense that if $f \overset{ext}{=} f'$, then $\mathsf{rep}(\mathsf{Next}\, f, l :: ls)$ iff $\mathsf{rep}(\mathsf{Next}\, f', l :: ls)$. Therefore, unlike the proofs in the previous section, the proofs of the following lemmas do not depend on extensionality principles.

The recursive function $\mathsf{extract}$, equationally specified in Section 3 as a continuation, is indeed an extractor for rep and the empty list:

▶ **Lemma 8.** $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$, *i. e.,* $\forall c, ls\,.\,\mathsf{rep}(c, ls) \to \mathsf{extract}\, c = \mathsf{flatten}\, ls$.

Proof. Setting $R_0(c, ls'') \overset{\text{Def}}{\equiv} \mathsf{extract}\, c = \mathsf{flatten}\, ls''$, $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$ is equivalent to $\mathsf{rep} \subseteq R_0$. We prove the latter by (non-strictly positive) induction, i. e., we show that the closure conditions (over) and (next) hold if rep is replaced by R_0.

(over): $R_0(\mathsf{Over}, [])$ means $\mathsf{extract}\,\mathsf{Over} = \mathsf{flatten}\,[]$, which holds since both sides equal $[]$.

(next): Assume $\forall k, ls'\,.\,\mathsf{isextractor}(R_0, ls', k) \to f\, k = l +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\, ls'\, ls)$, which is our induction hypothesis. Since, trivially, $\mathsf{isextractor}(R_0, [], \mathsf{extract})$, the induction hypothesis yields $f\,\mathsf{extract} = l +\!\!+ \mathsf{flatten}\, ls$, which is equivalent to our goal, $R_0(\mathsf{Next}\, f, l :: ls)$. ◀

The following lemma shows that $\mathsf{br}\,t$, defined in Section 2 as a routine transformer, is well-behaved w. r. t. representation: if the argument routine c represents a (double) list ls, then the resulting routine represents $\mathsf{zip}\,(\mathsf{niv}\,t)\,ls$: [7]

▶ **Lemma 9.** $\mathsf{rep}(c, ls) \to \mathsf{rep}(\mathsf{br}\,t\,c, \mathsf{zip}\,(\mathsf{niv}\,t)\,ls)$.

Proof. Induction on $t : \mathsf{Tree}$.

Case $t = \mathsf{Leaf}\,n$: Assume $\mathsf{rep}(c, ls)$.

We have to show $\mathsf{rep}(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,k), \mathsf{zip}\,[[n]]\,ls)$.

Subcase $ls = []$: Then $\mathsf{zip}\,[[n]]\,ls = [n] :: []$ and, since $\mathsf{rep}(c, [])$, $c = \mathsf{Over}$. Hence we have to show $\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,\mathsf{Over}\,k), [n] :: []\big)$, i. e., for all k, ls', if $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, then $n :: k\,\mathsf{Over} = [n] +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\,ls'\,[])$, i. e., $k\,\mathsf{Over} = \mathsf{flatten}\,ls'$. But the latter is obtained by instantiating the assumption $\mathsf{isextractor}(\mathsf{rep}, ls', k)$ with Over and $[]$.

Subcase $ls = l :: ls_0$: Then $\mathsf{zip}\,[[n]]\,ls = (n :: l) :: ls_0$ and, since $\mathsf{rep}(c, l :: ls_0)$, $c = \mathsf{Next}\,f$ with

$$(+) \qquad \forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\,k = l +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0)\ .$$

We have to show that $\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,(\mathsf{Next}\,f)\,k), (n :: l) :: ls_0\big)$, i. e.,

$$\forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to n :: f\,k = (n :: l) +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0)\ .$$

But, cancelling n, this is exactly $(+)$.

Case $t = \mathsf{Node}\,tl\,n\,tr$: By induction hypothesis, for all c, ls with $\mathsf{rep}(c, ls)$ and all $t' \in \{tl, tr\}$, $\mathsf{rep}(\mathsf{br}\,t'\,c, \mathsf{zip}\,(\mathsf{niv}\,t')\,ls)$.

Assume $\mathsf{rep}(c, ls)$. We have to show $\mathsf{rep}(\mathsf{br}\,t\,c, \mathsf{zip}\,(\mathsf{niv}\,t)\,ls)$, i. e.,

$$\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\ c\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)), \mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls\big)\ .$$

Subcase $ls = []$: Then $\mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls = [n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)$, and, since $\mathsf{rep}(c, [])$, $c = \mathsf{Over}$. Hence, we have to show that for all k, ls' such that $\mathsf{isextractor}(\mathsf{rep}, ls', k)$ we have $n :: (k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)\,\mathsf{Over} = [n] +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))$, i. e.,

$$k\,(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over})) = \mathsf{flatten}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))\ .$$

Using $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, instantiated with
$c := \mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over})$ and $ls'' := \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)$, our goal reduces to showing
$\mathsf{rep}\big(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,\mathsf{Over}), \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)\big)$ which, by the first induction hypothesis, further reduces to $\mathsf{rep}(\mathsf{br}\,tr\,\mathsf{Over}, \mathsf{niv}\,tr)$. Finally, by the second induction hypothesis (with $ls := []$), the latter reduces to (over).

Subcase $ls = l :: ls_0$: Then
$\mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls = (n :: l) :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0$ and therefore, by the assumption $\mathsf{rep}(c, ls)$, we get $c = \mathsf{Next}\,f$ with

$$(++) \qquad \forall k, ls'\,.\,\mathsf{isextractor}(\mathsf{rep}, ls', k) \to f\,k = l +\!\!+ \mathsf{flatten}\,(\mathsf{zip}\,ls'\,ls_0)\ .$$

We have to show
$\mathsf{rep}\big(\mathsf{Next}\,(\lambda k\,.\,n :: \mathsf{unfold}\,c\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr)), (n :: l) :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0\big)$,
i. e., for all k, ls' with $\mathsf{isextractor}(\mathsf{rep}, ls', k)$,

$$n :: f\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr) = (n :: l) +\!\!+ \mathsf{flatten}\big(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr))\,ls_0)\big)\ .$$

[7] This descriptional pattern suggests to define representation of double list transformers by routine transformers in the usual style of logical relations. With that definition in place, the lemma could be stated as representation of $\mathsf{zip}\,(\mathsf{niv}\,t)$ by $\mathsf{br}\,t$.

Deleting n and using associativity for zip we end up with the goal $f\,(k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr) = l +\!\!+ \mathsf{flatten}\big(\mathsf{zip}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))\,ls_0\big)$. By $(++)$ it suffices to show

$$\mathsf{isextractor}\big(\mathsf{rep}, \mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)), k \circ \mathsf{br}\,tl \circ \mathsf{br}\,tr\big).$$

Assume $\mathsf{rep}(c, ls'')$. We have to show

$$k\,(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,c)) = \mathsf{flatten}\,\big(\mathsf{zip}\,(\mathsf{zip}\,ls'\,(\mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{niv}\,tr)))\,ls''\big)\ .$$

By the assumption $\mathsf{isextractor}(\mathsf{rep}, ls', k)$, and using associativity of zip, it suffices to show $\mathsf{rep}\big(\mathsf{br}\,tl\,(\mathsf{br}\,tr\,c), \mathsf{zip}\,(\mathsf{niv}\,tl)\,(\mathsf{zip}\,(\mathsf{niv}\,tr)\,ls'')\big)$. The first induction hypothesis reduces this to $\mathsf{rep}(\mathsf{br}\,tr\,c, \mathsf{zip}\,(\mathsf{niv}\,tr)\,ls'')$ and the second further to $\mathsf{rep}(c, ls'')$, which holds by assumption. ◀

Alternative proof of Theorem 6. By the axiom (over), we have $\mathsf{rep}(\mathsf{Over}, [])$. Therefore, by Lemma 9, $\mathsf{rep}(\mathsf{br}\,t\,\mathsf{Over}, \mathsf{niv}\,t)$. Since, by Lemma 8, $\mathsf{isextractor}(\mathsf{rep}, [], \mathsf{extract})$, it follows $\mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) = \mathsf{flatten}\,(\mathsf{niv}\,t)$, i. e., $\mathsf{breadthfirst}\,t = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$. ◀

7 Verification by interpreting routines as recursive programs

In this section we give a correctness proof, which is based on understanding the elements of Rou as recursive programs. We give a meaning to routines by defining what it means for a routine to compute the breadth-first traversal of a tree, and use this in order to state and prove in Lemma 12 the correctness condition fulfilled by the key operation br.

Following Okasaki [13], one can understand the breadth-first traversal of a tree by understanding the more general notion of the breadth-first traversal of elements of $\mathsf{Forest} := \mathsf{List}\,\mathsf{Tree}$. We use ts (for lists of trees) as variables for forests.

The obvious lifting of $\mathsf{breadthfirst}_{\mathsf{spec}}$ to forests is

$$\mathsf{breadthfirst}_{\mathsf{f,spec}} \stackrel{\mathrm{Def}}{=} \mathsf{flatten} \circ \mathsf{niv}_{\mathsf{f}} : \mathsf{Forest} \to \mathsf{List}\,\mathbb{N}\ ,$$

where $\mathsf{niv}_{\mathsf{f}}$ zips all $\mathsf{niv}\,t$ for t in ts, i. e.

$$\mathsf{niv}_{\mathsf{f}} : \mathsf{Forest} \to \mathsf{List}^2\,\mathbb{N}$$
$$\mathsf{niv}_{\mathsf{f}}\,[] = [] \qquad \mathsf{niv}_{\mathsf{f}}\,(t :: ts) = \mathsf{zip}\,(\mathsf{niv}\,t)\,(\mathsf{niv}_{\mathsf{f}}\,ts)$$

Clearly, $\mathsf{breadthfirst}_{\mathsf{spec}}\,t = \mathsf{breadthfirst}_{\mathsf{f,spec}}\,[t]$.

It is our goal to prove the correctness of Hofmann's algorithm via an embedding of forests into routines that is in a certain sense simpler than the embedding γ and explains the roles of the functions $\mathsf{br} : \mathsf{Tree} \to \mathsf{Rou} \to \mathsf{Rou}$ and $\mathsf{extract} : \mathsf{Rou} \to \mathsf{List}\,\mathbb{N}$.

Our programs will not recurse on the length of a forest but on its depth, and will access its roots and its immediate subforest:

- $\mathsf{depth} : \mathsf{Tree} \to \mathbb{N}$, $\mathsf{depth}(\mathsf{Leaf}\,n) = 1$, $\mathsf{depth}(\mathsf{Node}\,tl\,n\,tr) = \max\{\mathsf{depth}\,tl\,,\,\mathsf{depth}\,tr\} + 1$.
- $\mathsf{depth}_{\mathsf{f}} : \mathsf{Forest} \to \mathbb{N}$, $\mathsf{depth}_{\mathsf{f}}\,[t_1, \ldots, t_n] = \max\{0, \mathsf{depth}\,t_1, \ldots, \mathsf{depth}\,t_n\}$.
- $\mathsf{roots} : \mathsf{Forest} \to \mathsf{List}\,\mathbb{N}$
  $\mathsf{roots}\,[] = []$ $\mathsf{roots}\,(\mathsf{Leaf}\,n :: ts) = \mathsf{roots}\,(\mathsf{Node}\,tl\,n\,tr :: ts) = n :: \mathsf{roots}\,ts$
- $\mathsf{sub} : \mathsf{Forest} \to \mathsf{Forest}$ calculates the immediate subforest:
  $\mathsf{sub}\,[] = [], \mathsf{sub}\,(\mathsf{Leaf}\,n :: ts) = \mathsf{sub}\,ts, \mathsf{sub}\,(\mathsf{Node}\,tl\,n\,tr :: ts) = tl :: tr :: \mathsf{sub}\,ts$.

▶ **Lemma 10.**
(a) $\mathsf{length}\,(\mathsf{niv_f}\ ts) = \mathsf{depth_f}\ ts$.
(b) *For* $ts \neq []$ *we have* $\mathsf{depth_f}\ ts = \mathsf{depth_f}\,(\mathsf{sub}\ ts) + 1$.
(c) *If* $ts \neq []$ *then* $\mathsf{niv_f}\ ts = \mathsf{roots}\ ts :: \mathsf{niv_f}\,(\mathsf{sub}\ ts)$.

Proof. Easy. ◀

We begin with the observation (which is made precise in Lemma 12 below) that the routines created in a run of the algorithm breadthfirst are either Over or of the form $(\mathsf{next}\,(\mathsf{addroots}\ ts)\ c)$ where

- $\mathsf{next} : (\mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}) \to \mathsf{Rou} \to \mathsf{Rou}$ $\quad \mathsf{next}\ g\ c = \mathsf{Next}\,(\lambda k\,.\,g\,(k\,c))$.
- $\mathsf{addroots} : \mathsf{Forest} \to \mathsf{List}\,\mathbb{N} \to \mathsf{List}\,\mathbb{N}$ $\quad \mathsf{addroots}\ ts = \mathsf{append}\,(\mathsf{roots}\ ts)$

We can regard these routines as recursive programs: Over is the routine which immediately terminates returning []. The routine $(\mathsf{next}\ g\ c)$ makes a recursive call to c, and if the result returned there is l it returns $(g\ l)$. extract executes these recursive programs: We have $\mathsf{extract}\ \mathsf{Over} = []$ and $\mathsf{extract}\,(\mathsf{next}\ g\ c) = g\,(\mathsf{extract}\ c)$.

We now construct for $ts : \mathsf{Forest}$ the routine $(\mathsf{c}\ ts)$ which represents the computation of the breadth-first traversal of ts. If $ts = []$, then Over represents the traversal of ts which is []. Otherwise, c represents the traversal of ts if it recursively calls a routine representing the traversal of $(\mathsf{sub}\ ts)$ and adds to the result $(\mathsf{roots}\ ts)$. More formally we define $\mathsf{c}\ ts : \mathsf{Rou}$ by recursion on the measure $\mathsf{depth_f}\ ts$:

$$\mathsf{c}\ ts = \begin{cases} \mathsf{Over} & \text{if } ts = [], \\ \mathsf{next}\,(\mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts)) & \text{otherwise.} \end{cases}$$

We show that extract evaluates the routines $\mathsf{c}\ ts$ to the breadth-first traversal of ts:

▶ **Lemma 11.** $\mathsf{extract} \circ \mathsf{c} \stackrel{ext}{=} \mathsf{breadthfirst_{f,spec}}$.

Proof. We show $\mathsf{extract}\,(\mathsf{c}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts$ by induction on $\mathsf{depth_f}\ ts$:
If $\mathsf{depth_f}\ ts = 0$ then $ts = []$, and $\mathsf{extract}\,(\mathsf{c}\ ts) = [] = \mathsf{flatten}\,(\mathsf{niv_f}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts$.
Otherwise by IH $\mathsf{extract}\,(\mathsf{c}\,(\mathsf{sub}\ ts)) = \mathsf{breadthfirst_{f,spec}}\,(\mathsf{sub}\ ts))$, and therefore, by Lemma 10

$$\begin{aligned} \mathsf{extract}\,(\mathsf{c}\ ts) &= \mathsf{extract}\,(\mathsf{next}\,(\mathsf{addroots}\ ts)\,(\mathsf{c}\,(\mathsf{sub}\ ts))) = \mathsf{addroots}\ ts\,(\mathsf{extract}\,(\mathsf{c}\,(\mathsf{sub}\ ts))) \\ &= \mathsf{roots}\ ts +\!\!+ \mathsf{flatten}\,(\mathsf{niv_f}\,(\mathsf{sub}\ ts)) = \mathsf{flatten}\,(\mathsf{roots}\ ts :: \mathsf{niv_f}\,(\mathsf{sub}\ ts)) \\ &= \mathsf{flatten}\,(\mathsf{niv_f}\ ts) = \mathsf{breadthfirst_{f,spec}}\ ts\ . \end{aligned}$$ ◀

The next lemma is a key lemma for br. It shows that $(\mathsf{br}\ t\ c)$ translates a routine c computing the traversal of ts into a routine computing the traversal of $(t :: ts)$:

▶ **Lemma 12.** $\mathsf{br}\ t\ \circ \mathsf{c} \stackrel{ext}{=} \mathsf{c} \circ \mathsf{cons}\ t$.

Proof. We show $\mathsf{br}\ t\,(\mathsf{c}\ ts) = \mathsf{c}\,(t :: ts)$ by induction on $\mathsf{depth}\ t$:
Case 1 $ts = []$. Then $\mathsf{c}\ ts = \mathsf{Over}$.
Case 1.1 $t = \mathsf{Leaf}\ n$. We have

$$\begin{aligned} \mathsf{br}\ t\,(\mathsf{c}\ ts) &= \mathsf{next}\,(\mathsf{cons}\ n)\,\mathsf{Over} \\ &= \mathsf{next}\,(\mathsf{addroots}\,(t :: ts))\,(\mathsf{c}\,(\mathsf{sub}\,(t :: ts))) = \mathsf{c}\,(t :: ts) \end{aligned}$$.

Case 1.2 $t = \mathsf{Node}\ tl\ n\ tr$. Then by IH we get

$$\begin{aligned} \mathsf{br}\ t\,(\mathsf{c}\ ts) &= \mathsf{next}\,(\mathsf{cons}\ n)\,(\mathsf{br}\ tl\,(\mathsf{br}\ tl\,(\mathsf{c}\ ts))) \\ &= \mathsf{next}\,(\mathsf{cons}\ n)\,(\mathsf{c}\,(tl :: tr :: ts)) \\ &= \mathsf{next}\,(\mathsf{addroots}\,(t :: ts))\,(\mathsf{c}\,(\mathsf{sub}\,(t :: ts))) = \mathsf{c}\,(t :: ts) \end{aligned}$$

Case 2 Otherwise. Then $\mathsf{c}\ ts = \mathsf{next}\ (\mathsf{addroots}\ ts)\ (\mathsf{c}\ (\mathsf{sub}\ ts))$.

Case 2.1 $t = \mathsf{Leaf}\ n$.

$$\begin{aligned}\mathsf{br}\ t\ (\mathsf{c}\ ts) &= \mathsf{next}\ (\mathsf{cons}\ n \circ \mathsf{addroots}\ ts)\ (\mathsf{c}\ (\mathsf{sub}\ ts))\\ &= \mathsf{next}\ (\mathsf{addroots}\ (t :: ts))\ (\mathsf{c}\ (\mathsf{sub}\ (t :: ts))) = \mathsf{c}\ (t :: ts)\end{aligned}$$

Case 2.2 $t = \mathsf{Node}\ tl\ n\ tl$. Then

$$\begin{aligned}\mathsf{br}\ t\ (\mathsf{c}\ ts) &= \mathsf{next}\ (\mathsf{cons}\ n \circ \mathsf{addroots}\ ts)\ (\mathsf{br}\ tl\ (\mathsf{br}\ tl\ (\mathsf{c}\ (\mathsf{sub}\ ts))))\\ &= \mathsf{next}\ (\mathsf{addroots}\ (t :: ts))\ (\mathsf{c}\ (tl :: tr :: (\mathsf{sub}\ ts)))\\ &= \mathsf{next}\ (\mathsf{addroots}\ (t :: ts))\ (\mathsf{c}\ (\mathsf{sub}\ (t :: ts))) = \mathsf{c}\ (t :: ts)\end{aligned}$$

◀

Alternative proof of Theorem 6. $\mathsf{breadthfirst}\ t = \mathsf{extract}\ (\mathsf{br}\ t\ \mathsf{Over}) = \mathsf{extract}\ (\mathsf{br}\ t\ (\mathsf{c}\ [])) = \mathsf{extract}\ (\mathsf{c}\ [t]) = \mathsf{breadthfirst}_{\mathsf{f,spec}}\ [t] = \mathsf{breadthfirst}_{\mathsf{spec}}\ t$. ◀

8 A predicative version of breadthfirst

In this section we present a variant of breadth-first traversal that, like Hofmann's algorithm, avoids the repeated use of list concatenation but is predicative since it doesn't use the data type of routines. Instead lists of functions are used as intermediate data type.

As observed in the previous section, the only elements of Rou created by the operations br and breadthfirst are Over and $\mathsf{next}\ g\ c$, where $g : \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N}$ and $c : \mathsf{Rou}$, and c is itself created by the algorithm. We can represent the elements of Rou that are defined inductively by these clauses as lists of functions $g : \mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N}$, and therefore obtain them as those in the image of the function Φ defined as follows:

$$\mathsf{Rou}' = \mathsf{List}(\mathsf{List}\ \mathbb{N} \to \mathsf{List}\ \mathbb{N})$$
$$\Phi : \mathsf{Rou}' \to \mathsf{Rou} \qquad \Phi\ [] = \mathsf{Over} \qquad \Phi\ (g :: gs) = \mathsf{next}\ g\ (\Phi\ gs)$$

We denote elements of Rou′ with the variable gs.

We translate br into a function br′ referring to Rou′ s. t. $\Phi \circ \mathsf{br}'\ t \overset{ext}{=} \mathsf{br}\ t \circ \Phi$:

$$\begin{aligned}&\mathsf{br}' : \mathsf{Tree} \to \mathsf{Rou}' \to \mathsf{Rou}'\\ &\mathsf{br}'\ (\mathsf{Leaf}\ n)\ [] &&= \mathsf{cons}\ n :: []\\ &\mathsf{br}'\ (\mathsf{Leaf}\ n)\ (g :: gs) &&= (\mathsf{cons}\ n \circ g) :: gs\\ &\mathsf{br}'\ (\mathsf{Node}\ tl\ n\ tr)\ [] &&= \mathsf{cons}\ n :: \mathsf{br}'\ tl\ (\mathsf{br}'\ tr\ [])\\ &\mathsf{br}'\ (\mathsf{Node}\ tl\ n\ tr)\ (g :: gs) &&= (\mathsf{cons}\ n \circ g) :: \mathsf{br}'\ tl\ (\mathsf{br}'\ tr\ gs)\end{aligned}$$

The defining equations for br′ are easily derived by transforming the right-hand side of the desired functional equation $\Phi\ (\mathsf{br}'\ t\ gs) = \mathsf{br}\ t\ (\Phi\ gs)$ into the form $\Phi\ gs'$ and then setting $\mathsf{br}'\ t\ gs = gs'$.

▶ **Lemma 13.** $\Phi \circ \mathsf{br}'\ t \overset{ext}{=} \mathsf{br}\ t \circ \Phi$.

Proof. One shows $\Phi\ (\mathsf{br}'\ t\ gs) = \mathsf{br}\ t\ (\Phi\ gs)$ by a straightforward induction on t and case analysis on gs (formalized in the Coq proof `br'_Lemma`, see Section 11). ◀

We can in the same way translate extract into a function extract′ operating on Rou′ s. t. $\mathsf{extract}' \overset{ext}{=} \mathsf{extract} \circ \Phi$: From this condition one can immediately derive its defining equations:

$$\mathsf{extract}' : \mathsf{Rou}' \to \mathsf{List}\ \mathbb{N} \qquad \mathsf{extract}'\ [] = [] \qquad \mathsf{extract}'\ (g :: gs) = g\ (\mathsf{extract}'\ gs)$$

▶ **Lemma 14.** $\mathsf{extract}' \overset{ext}{=} \mathsf{extract} \circ \Phi$.

Proof. We show $\mathsf{extract}'\ gs = \mathsf{extract}\,(\Phi\ gs)$ by induction on gs:

$$\begin{array}{lcl}\mathsf{extract}'\,[] & = & [] = \mathsf{extract}\,(\Phi\,[])\\ \mathsf{extract}'\,(g :: gs) & = & g\,(\mathsf{extract}'\ gs) = g\,(\mathsf{extract}\,(\Phi\ gs))\\ & = & \mathsf{extract}\,(\mathsf{next}\,g\,(\Phi\ gs)) = \mathsf{extract}\,(\Phi\,(g :: gs))\end{array}$$

◀

Now we define $\mathsf{breadthfirst}' : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$, $\mathsf{breadthfirst}'\,t = \mathsf{extract}'\,(\mathsf{br}'\,t\,[])$. It follows:

▶ **Lemma 15.** $\mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}$.

Proof. $\mathsf{breadthfirst}'\,t = \mathsf{extract}'\,(\mathsf{br}'\,t\,[]) = \mathsf{extract}\,(\Phi\,(\mathsf{br}'\,t\,[])) = \mathsf{extract}\,(\mathsf{br}\,t\,(\Phi\,[]))$
$= \mathsf{extract}\,(\mathsf{br}\,t\,\mathsf{Over}) = \mathsf{breadthfirst}\,t.$ ◀

In the next section 9 we will see how $\mathsf{breadthfirst}'$ can be reduced to $\mathsf{breadthfirst}''$ which is extensionally equal to $\mathsf{breadthfirst}_{\mathsf{spec}}$, giving an algebraic proof of the correctness of $\mathsf{breadthfirst}$. However, we can give as well a direct correctness proof of $\mathsf{breadthfirst}'$:
The routine computing the traversal of a $ts : \mathsf{Forest}$ having $\mathsf{niv}_{\mathsf{f}} = [l_1, \ldots, l_m]$ is given by $\mathsf{traverse}\,ts = [\mathsf{append}\,l_1, \ldots, \mathsf{append}\,l_n]$. A recursive definition (recursion on the measure $\mathsf{depth}\,ts$) of $\mathsf{traverse}\,ts : \mathsf{Rou}'$ is as follows:

$$\mathsf{traverse}\,ts = \begin{cases} [] & \text{if } ts = [],\\ \mathsf{addroots}\,ts :: \mathsf{traverse}\,(\mathsf{sub}\,ts) & \text{otherwise.}\end{cases}$$

▶ **Lemma 16.** $\mathsf{extract}' \circ \mathsf{traverse} \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{f,spec}}$.

▶ **Lemma 17.** $\mathsf{br}'\,t \circ \mathsf{traverse} \overset{ext}{=} \mathsf{traverse} \circ (\mathsf{cons}\,t)$.

Proof of Lemmas 16 and 17. One shows $\mathsf{extract}'\,(\mathsf{traverse}\ ts) = \mathsf{breadthfirst}_{\mathsf{f,spec}}\,ts$ by induction on $\mathsf{depth}\,ts$ and $\mathsf{br}'\,t\,(\mathsf{traverse}\,ts) = \mathsf{traverse}\,(t :: ts)$ by induction on t. ◀

We obtain an **alternative proof of Theorem 6** which contains as well the correctness of $\mathsf{breadthfirst}'$:

▶ **Theorem 18.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$.

Proof. The first equation is Lemma 15. The 2nd equation follows as the alternative proof of Theorem 6 in Sect. 7 but using Lemmas 16 and 17 instead of Lemmas 11 and 12, respectively, and replacing Over by $[] : \mathsf{Rou}'$. ◀

9 A simplified predicative version of breadthfirst

The predicative algorithm for breadth-first traversal developed in the previous section can be simplified by observing that the type Rou' is only used with lists of functions that are formed from $(\mathsf{cons}\,n)$ by composition, i. e., functions of the form $\lambda l\,.\,l' +\!\!+ l$ for some $l' : \mathsf{List}\,\mathbb{N}$. We can therefore denote them by elements of $\mathsf{List}\,\mathbb{N}$, and the elements of Rou' by elements of $\mathsf{List}^2\,\mathbb{N}$. Therefore, we define

$$\begin{array}{ll}\mathsf{Rou}'' := \mathsf{List}^2\,\mathbb{N} & \\ \Psi : \mathsf{Rou}'' \to \mathsf{Rou}' & \Psi\ ls = \mathsf{map}\ \mathsf{append}\ ls\\ \text{where}\quad \mathsf{map} : (A \to B) \to \mathsf{List}\,A \to \mathsf{List}\,B & \mathsf{map}\,f\,[l_1, \ldots, l_n] = [f\,l_1, \ldots, f\,l_n]\end{array}$$

We translate br' into a function br'' referring to Rou'':

$$\begin{array}{lcl}\mathsf{br}'' : \mathsf{Tree} \to \mathsf{Rou}'' \to \mathsf{Rou}'' & & \\ \mathsf{br}''\,(\mathsf{Leaf}\,n)\,[] & = & [[n]]\\ \mathsf{br}''\,(\mathsf{Leaf}\,n)\,(l :: ls) & = & \mathsf{cons}\,n\,l :: ls\\ \mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,[] & = & [n] :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,[])\\ \mathsf{br}''\,(\mathsf{Node}\,tl\,n\,tr)\,(l :: ls) & = & \mathsf{cons}\,n\,l :: \mathsf{br}''\,tl\,(\mathsf{br}''\,tr\,ls)\end{array}$$

▶ **Lemma 19.** $\Psi \circ \mathsf{br}''\, t \overset{ext}{=} \mathsf{br}'\, t \circ \Psi$.

Proof. We show $\Psi\,(\mathsf{br}''\, t\, ls) = \mathsf{br}'\, t\,(\Psi\, ls)$ by induction on t:

$$\begin{aligned}
\Psi\,(\mathsf{br}''\,(\mathsf{Leaf}\, n)\,[]) &= \Psi\,[[n]] = \mathsf{cons}\, n :: [] = \mathsf{br}'\,(\mathsf{Leaf}\, n)\,[]\\
\Psi\,(\mathsf{br}''\,(\mathsf{Leaf}\, n)\,(l :: ls)) &= \Psi\,(\mathsf{cons}\, n\, l :: ls)\\
&= (\mathsf{cons}\, n \circ \mathsf{append}\, l) :: \Psi\, ls\\
&= \mathsf{br}'\,(\mathsf{Leaf}\, n)\,(\mathsf{append}\, l :: \Psi\, ls)\\
\Psi\,(\mathsf{br}''\,(\mathsf{Node}\, tl\, n\, tr)\,[]) &= \Psi\,([n] :: \mathsf{br}''\, tl\,(\mathsf{br}''\, tr\,[]))\\
&= \mathsf{cons}\, n :: \mathsf{br}'\, tl\,(\mathsf{br}'\, tr\,[])\\
&= \mathsf{br}'\,(\mathsf{Node}\, tl\, n\, tr)\,[]\\
\Psi\,(\mathsf{br}''\,(\mathsf{Node}\, tl\, n\, tr)\,(l :: ls)) &= \Psi(\mathsf{cons}\, n\, l :: (\mathsf{br}''\, tl\,(\mathsf{br}''\, tr\, ls)))\\
&= \mathsf{cons}\, n \circ \mathsf{append}\, l :: (\mathsf{br}'\, tl\,(\mathsf{br}'\, tr\,(\Psi\, ls)))\\
&= \mathsf{br}'\,(\mathsf{Node}\, tl\, n\, tr)\,(\mathsf{append}\, l :: \Psi\, ls)
\end{aligned}$$

◀

▶ **Lemma 20.** $\mathsf{br}''\, t \overset{ext}{=} \mathsf{zip}\,(\mathsf{niv}\, t)$.

Proof. We show $\mathsf{br}''\, t\, ls = \mathsf{zip}\,(\mathsf{niv}\, t)\, ls$ by induction on t: For $t = \mathsf{Leaf}\, n$ this follows immediately by the definition of br''. In the case that $t = \mathsf{Node}\, tl\, n\, tr$ and $ls = []$ we get using the IH $\mathsf{br}''\, t\, ls = [n] :: \mathsf{br}''\, tl\,(\mathsf{br}''\, tr\,[]) = [n] :: \mathsf{zip}\,(\mathsf{niv}\, tl)\,(\mathsf{zip}\,(\mathsf{niv}\, tr)\,[]) = [n] :: \mathsf{zip}\,(\mathsf{niv}\, tl)\,(\mathsf{niv}\, tr) = \mathsf{niv}\, t = \mathsf{zip}\,(\mathsf{niv}\, t)\,[]$. In case of $t = \mathsf{Node}\, tl\, n\, tr$ and $ls = l' :: ls'$ we get using the IH $\mathsf{br}''\, t\, ls = \mathsf{cons}\, n\, l' :: \mathsf{br}''\, tl\,(\mathsf{br}''\, tr\, ls') = \mathsf{cons}\, n\, l' :: \mathsf{zip}\,(\mathsf{niv}\, tl)\,(\mathsf{zip}\,(\mathsf{niv}\, tr)\, ls') = \mathsf{cons}\, n\, l' :: \mathsf{zip}\,(\mathsf{zip}\,(\mathsf{niv}\, tl)\,(\mathsf{zip}\,(\mathsf{niv}\, tr)))\, ls') = \mathsf{zip}\,([n] :: \mathsf{zip}\,(\mathsf{niv}\, tl)\,(\mathsf{zip}\,(\mathsf{niv}\, tr)))\,(l' :: ls') = \mathsf{zip}(\mathsf{niv}\, t)\, ls$. ◀

▶ **Lemma 21.** $\mathsf{flatten} \overset{ext}{=} \mathsf{extract}' \circ \Psi$.

Proof. By induction on the list argument:

$$\begin{aligned}
\mathsf{flatten}\,[] &= [] = \mathsf{extract}'\,[]\\
\mathsf{flatten}\,(l :: ls) &= l +\!\!+ \mathsf{flatten}\, ls = \mathsf{append}\, l\,(\mathsf{extract}'\,(\Psi\, ls)) = \mathsf{extract}'\,(\Psi\,(l :: ls))
\end{aligned}$$

◀

Now we define $\mathsf{breadthfirst}'' : \mathsf{Tree} \to \mathsf{List}\,\mathbb{N}$ by $\mathsf{breadthfirst}''\, t = \mathsf{flatten}\,(\mathsf{br}''\, t\,[])$.

We obtain an **alternative proof of Theorem 6** which contains as well the correctness of $\mathsf{breadthfirst}'$ and $\mathsf{breadthfirst}''$:

▶ **Theorem 22.** $\mathsf{breadthfirst} \overset{ext}{=} \mathsf{breadthfirst}' \overset{ext}{=} \mathsf{breadthfirst}'' \overset{ext}{=} \mathsf{breadthfirst}_{\mathsf{spec}}$.

Proof. The first equation is Lemma 15. We prove the second equation:
$\mathsf{breadthfirst}''\, t = \mathsf{flatten}\,(\mathsf{br}''\, t\,[]) = \mathsf{extract}'\,(\Psi\,(\mathsf{br}''\, t\,[])) = \mathsf{extract}'\,(\mathsf{br}'\, t\,(\Psi\,[])) = \mathsf{extract}'\,(\mathsf{br}'\, t\,[]) = \mathsf{breadthfirst}'\, t$.
Furthermore, by Lemma 20, we get
$\mathsf{breadthfirst}''\, t = \mathsf{flatten}\,(\mathsf{br}''\, t\,[]) = \mathsf{flatten}\,(\mathsf{zip}\,(\mathsf{niv}\, t)\,[]) = \mathsf{flatten}\,(\mathsf{niv}\, t) = \mathsf{breadthfirst}_{\mathsf{spec}}\, t$. ◀

10 Formal comparison of the obtained algorithms and proofs

In this section we isolate the common structure of the algorithms and proofs we have seen so far. Since, as remarked earlier, breadth-first traversal is not modular, all algorithms first compute some intermediate result (in a modular way) from which then the final result can be easily extracted. In fact, the program computing the intermediate result has an extra parameter which makes it possible to replace list concatenation (featuring in the specification) by function composition. We capture this common structure by the notion of a "system" and show that all proofs boil down to establishing a "simulation" relation between systems.

▶ **Definition 23.**

- *A* system *is a quadruple* $S = (A, \mathsf{Nil}, g, e)$ *where* $A : \mathsf{Set}$, $\mathsf{Nil} : A$, $g : \mathsf{Tree} \to A \to A$, *and* $e : A \to \mathsf{List}\,\mathbb{N}$.
- S *is* correct *(for breadth-first traversal) if* $e\,(g\,t\,\mathsf{Nil}) = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$ *for all trees* t.
- *Let* $S' = (A', \mathsf{Nil}', g', e')$ *be another system. A relation* R *on* $A \times A'$ *is a* simulation *between* S *and* S', $S \overset{R}{\sim} S'$, *if (1)* $R(\mathsf{Nil}, \mathsf{Nil}')$, *and, whenever* $R(a, a')$, *then (2)* $R(g\,t\,a, g'\,t\,a')$ *for all trees* t, *and (3)* $e\,a = e'\,a'$.
- *Let* S, S' *be systems.* S *and* S' *are* similar, $S \sim S'$, *if there exists a simulation between* S *and* S'.

▶ **Lemma 24.** *If* $S \sim S'$ *then* S *is correct if and only if* S' *is correct.*

Proof. If $S \overset{R}{\sim} S'$, then $R(g\,t\,\mathsf{Nil}, g'\,t\,\mathsf{Nil}')$, by (1) and (2), hence $e\,(g\,t\,\mathsf{Nil}) = e'\,(g'\,t\,\mathsf{Nil}')$, by (3). ◀

Note that if R is *functional*, i. e., defined as the graph of a function $\phi : A' \to A$, by setting $R(a, a')$ iff $a = \phi\,a'$, then the simulation conditions become (1) $\mathsf{Nil} = \phi\,\mathsf{Nil}'$, (2) $g\,t \circ \phi \overset{ext}{=} \phi \circ g'\,t$ for all trees t, and (3) $e \circ \phi \overset{ext}{=} e'$. In this situation we write $S \overset{\phi}{\leftarrow} S'$. All but one of the simulations described below are functional.

The specification of breadth-first traversal given in Section 2 corresponds to the system $S_{\mathsf{spec}} \overset{\mathrm{Def}}{=} (\mathsf{List}^2\mathbb{N}, [], \mathsf{zip} \circ \mathsf{niv}, \mathsf{flatten})$. Correctness holds since $\mathsf{flatten}\,((\mathsf{zip} \circ \mathsf{niv})\,t\,[]) = \mathsf{flatten}\,(\mathsf{niv}\,t) = \mathsf{breadthfirst}_{\mathsf{spec}}\,t$.

In the new view of systems, we may say that Hofmann defined his algorithm $\mathsf{breadthfirst}$ by the system $S_{\mathsf{MH}} \overset{\mathrm{Def}}{=} (\mathsf{Rou}, \mathsf{Over}, \mathsf{br}, \mathsf{extract})$ (Sect. 3) and showed that $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{Over}}}{\leftarrow} S_{\mathsf{spec}}$ where $\gamma_{\mathsf{Over}}\,ls \overset{\mathrm{Def}}{=} \gamma\,ls\,\mathsf{Over}$ (Sect. 4). Condition (1) holds by the definition of γ, (2) holds by Lemmas 4 and 5, and (3) is Lemma 3.

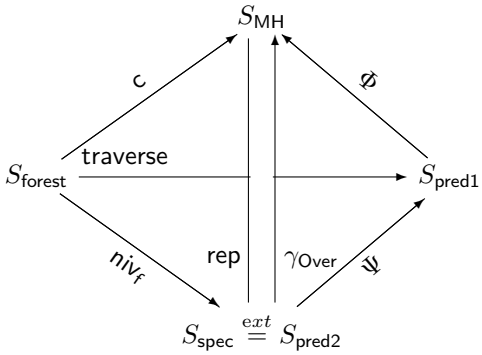
The proofs given in Section 6 amount to showing $S_{\mathsf{MH}} \overset{\mathsf{rep}}{\sim} S_{\mathsf{spec}}$. (1) is the axiom (over), (2) is Lemma 9, and (3) is Lemma 8.

The (spec.-like) algorithm $\lambda t\,.\,\mathsf{breadthfirst}_{\mathsf{f,spec}}\,[t]$ of Section 7 works with forests as the intermediate data type. The underlying system is $S_{\mathsf{forest}} \overset{\mathrm{Def}}{=} (\mathsf{Forest}, [], \mathsf{cons}, \mathsf{breadthfirst}_{\mathsf{f,spec}})$. Correctness of this system is easily established via the functional simulation $S_{\mathsf{spec}} \overset{\mathsf{niv_f}}{\leftarrow} S_{\mathsf{forest}}$ ((2) holds by definition of $\mathsf{niv_f}$, (3) is trivial). However, the point of S_{forest} is to provide a new correctness proof for S_{MH}. This is achieved by showing $S_{\mathsf{MH}} \overset{\mathsf{c}}{\leftarrow} S_{\mathsf{forest}}$. (1) holds by definition of c, (2) is Lemma 12, and (3) is Lemma 11.

The first predicative version of breadth-first traversal introduced in Section 8 defines the system $S_{\mathsf{pred1}} \overset{\mathrm{Def}}{=} (\mathsf{Rou}', [], \mathsf{br}', \mathsf{extract}')$ and proves the simulation $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}}$. The simulation conditions (2),(3) are shown in Lemmas 13 and 14, while (1) holds by definition of Φ. The correctness of S_{pred1} is shown via the simulation $S_{\mathsf{pred1}} \overset{\mathsf{traverse}}{\leftarrow} S_{\mathsf{forest}}$.

The simplified predicative algorithm in Section 9 is defined by the system $S_{\mathsf{pred2}} \overset{\mathrm{Def}}{=} (\mathsf{List}^2\mathbb{N}, [], \mathsf{br}'', \mathsf{flatten})$. S_{pred2} is in fact (extensionally) equal to S_{spec} since $\mathsf{br}'' \overset{ext}{=} \mathsf{zip} \circ \mathsf{niv}$, by Lemma 20. We show $S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$: the simulation conditions (2),(3) are given by the Lemmas 19 and 21, while (1) holds by definition of Ψ.

The following diagram gives an overview of the simulations:



In fact, the functions in the diagram are fully commutative assuming extensionality (regarding rep all we know at this stage is that it is a simulation, but we don't know its relationship to the simulation defined by γ_{Over}):

▶ **Lemma 25.**
(a) $\gamma_{\mathsf{Over}} \overset{ext}{=} \Phi \circ \Psi$.
(b) $\mathsf{traverse} \overset{ext}{=} \Psi \circ \mathsf{niv_f}$.
(c) $\mathsf{c} \overset{ext}{=} \Phi \circ \mathsf{traverse} \overset{ext}{=} \gamma_{\mathsf{Over}} \circ \mathsf{niv_f}$.

Proof. $\Phi\,(\Psi\,ls) = \gamma_{\mathsf{Over}}\,ls$ can be easily shown by induction on ls. However, the proof uses the extensionality principle (cf. Section 4). The equation $\mathsf{traverse}\,ts = \Psi(\mathsf{niv_f}\,ts)$ is obvious from the definition of traverse. $\mathsf{c}\,ts = \Phi\,(\mathsf{traverse}\,ts)$ follows by induction on depth ts. $\mathsf{c} \overset{ext}{=} \gamma_{\mathsf{Over}} \circ \mathsf{niv_f}$ follows from the previous equations. ◀

In particular, the simulations $S_{\mathsf{MH}} \overset{\Phi}{\leftarrow} S_{\mathsf{pred1}} \overset{\Psi}{\leftarrow} S_{\mathsf{pred2}}$ provide a splitting of Hofmann's simulation $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{Over}}}{\leftarrow} S_{\mathsf{spec}}$ into simpler components.

11 Implementation and formalization in proof assistants

Here, we comment on our (partial) implementation of the presented ideas in Coq and Agda, that is publicly available in a Git repository [2]. The *Coq system* does not allow any inductive data type beyond strictly positive ones.[8] We overcome this by working with a version of Coq augmented by the plugin `TypingFlags` provided by Simon Boulier.[9] The effect of this plugin is to disable the checks for strict positivity, guardedness and termination. If, in such a development, one has established Lemma `lem` (for example), then `Print Assumptions lem` reveals for which constructions the plugin has forced Coq to accept them. For the formalization of Theorem 6, the forced acceptance only concerns the inductive data type Rou and the recursive function extract (and we also referred to `Logic.FunctionalExtensionality.functional_extensionality`, which is nothing but assuming equality of pointwise equal functions). The formalization and its verification present no difficulties at all, given the detailed proofs we provide in the paper. Thus, all of the elaborated mathematical developments in the Sections 2 to 10, with the notable exception of Section 5 (that is situated outside of Coq since it reflects on the term evaluation mechanism)

[8] See the Coq reference manual, in particular `https://coq.inria.fr/distrib/current/refman/language/cic.html#positivity-condition`.
[9] Plugin available at `https://github.com/SimonBoulier/TypingFlags/`.

are fully formalized in Coq, under the above provisos, i. e., with forced acceptance by Coq of the type Rou, the function extract, the relation rep and its induction principle `rep_ind` that is "manually" defined and not generated by the system, and by sometimes employing extensionality. For the recapitulations in form of the four formalized correctness proofs of S_{MH} – through Hofmann's function γ, through the relation rep, through forests and through the two predicative systems, lines of the form `Print Assumptions S_MH_correct*` reveal what is assumed beyond the core of Coq: Rou and extract in all cases since the algorithm is expressed in terms of them, rep and its induction principle only for the second proof, and extensionality only for the first and fourth proof.

Agda has the feature that using pragmas one can switch off strict positivity checks locally for data types and termination checks locally for functions. This allowed us to implement the functions used in the paper. Using quantification of set levels we were able to write down a substantial part of the operations defined in System F in Sect. 5, and after using postulates and the REWRITE pragma as well the extension by Mendler recursion. This allowed us to check that the reductions hold (at least that the left-hand and right-hand side of a reduction have the same normal form). Carrying out the proofs not requiring extensionality is still work in progress.

12 Conclusion and further work

In this paper we studied an intriguing algorithm by Martin Hofmann for the breadth-first traversal of finite binary trees which uses a non-strictly positive data type Rou of routines. We completed Hofmann's proof sketch of correctness (Sect. 4) and provided a justification for the termination of the algorithm by reduction to Mendler-style recursion in system F (Sect. 5). Furthermore we presented various alternative breadth-first traversal algorithms and correctness proofs with the aim to provide an explanation of Hofmann's somewhat mysterious construction. In Sect. 6 we transformed the data type Rou into a non-strictly positive inductive relation rep between routines and double lists and proved directly that the algorithm maps a tree to a routine that represents its levels from which correctness follows immediately. While the proof in Sect. 6 exploits non-strict positive induction as a proof principle, the other proofs only use structural induction (on lists or trees) but instead introduce new constructions that explain the roles of the components of Hofmann's algorithm and break it (the algorithm) into smaller, simpler, parts. The proof in Sect. 7 proves the correctness of Hofmann's algorithm breadthfirst via a simulation by a straightforward extension of breadth-first traversal to forests (which is closely related to the common approach to breadth-first traversal [13]). This reveals that the crucial component, br, of breadthfirst performs – via this simulation – nothing but the cons-operation on lists of trees. Through an analysis of the behaviour of breadthfirst we showed in Section 8 how to replace the impredicative type Rou of routines by the type Rou′ of lists of list functions and provided a predicative version, breadthfirst′, of breadthfirst. In Section 9, this predicative algorithm is further simplified by observing that only functions of the form $\lambda l\,.\,l' +\!\!+ l$ are needed which can be represented by the list l'. Section 10 isolates the common structure of the algorithms by the notion of a *system* and the common structure of the correctness proofs by the notion of a *simulation*. In addition it shows that the simulation $S_{\mathsf{MH}} \overset{\gamma_{\mathsf{over}}}{\longleftarrow} S_{\mathsf{spec}}$, which corresponds to Hofmann's original proof, is split into the two, simpler and predicative, simulations $S_{\mathsf{MH}} \overset{\Phi}{\longleftarrow} S_{\mathsf{pred1}} \overset{\Psi}{\longleftarrow} S_{\mathsf{pred2}}$.

All algorithms were implemented and verified in the proof assistant Coq using various tweaks and extensions to accommodate non-strict positivity and some algorithms were implemented in Agda and Haskell [2].

Is the mystery of non-strictly positive breadth-first traversal now completely solved? Far from it. Looking at the algorithms it is quite clear that they should work for infinite (and hence non-well-founded) binary trees as well. This is confirmed by experiments with implementations in Haskell [2]. In order to formally prove this, coinductive data types and proof principles will be required which rely on the productivity of algorithms instead of the well-foundedness of their inputs. Carrying this out in current proof systems (whose capabilities of dealing with coinduction are still in their infancy) will be an exciting challenge.

Another mysterious algorithm that can be formulated with a non-strictly positive inductive type similar to the type of routines is a solution to the "same-fringe problem" that was suggested to us by Olivier Danvy. The problem is well-known: testing whether two finite trees have the same fringe, i. e., the same left-to-right listing of labels at their leaves. This problem is essentially different from breadth-first traversal since it relies on trees being finite. Its analysis is left to further work.

References

1. Andreas Abel and Ralph Matthes. Fixed Points of Type Constructors and Primitive Recursion. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic, 18th International Workshop, CSL 2004, Karpacz, Poland, Proceedings*, volume 3210 of *Lecture Notes in Computer Science*, pages 190–204. Springer, 2004. doi:10.1007/978-3-540-30124-0_17.
2. Ulrich Berger, Ralph Matthes, and Anton Setzer. Git repository of code supplementing the present paper. https://github.com/rmatthes/breadthfirstalahofmann, 2018–2019.
3. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de Doctorat d'État, Université de Paris VII, 1972.
4. Jean-Yves Girard. *Proofs and types*. Cambridge Tracts in Theoretical Computer Science 7. Cambridge University Press, Cambridge, 1989.
5. Robert Harper and John C. Mitchell. Parametricity and variants of Girard's J operator. *Information Processing Letters*, 70:1–5, 1999.
6. Martin Hofmann. Non Strictly Positive Datatypes in System F, February 1993. Email on types mailing list, http://www.seas.upenn.edu/~sweirich/types/archive/1993/msg00027.html.
7. Martin Hofmann. Approaches to Recursive Datatypes – a Case Study, April 1995. LaTeX draft, 5 pages. Circulated by email.
8. Geraint Jones and Jeremy Gibbons. Linear-time Breadth-first Tree Algorithms: An Exercise in the Arithmetic of Folds and Zips. Technical Report No. 71, Dept of Computer Science, University of Auckland, 1993. IFIP Working Group 2.1 working paper 705 WIN-2. URL: http://www.cs.ox.ac.uk/people/jeremy.gibbons/publications/linear.ps.gz.
9. Ralph Matthes. *Extensions of System F by Iteration and Primitive Recursion on Monotone Inductive Types*. Doktorarbeit (PhD thesis), LMU München, 1998. Available via the homepage http://www.irit.fr/~Ralph.Matthes/works.html.
10. Ralph Matthes. Tarski's fixed-point theorem and lambda calculi with monotone inductive types. *Synthese*, 133(1):107–129, 2002.
11. Paul F. Mendler. Inductive Definition in Type Theory. Technical Report 87-870, Cornell University, Ithaca, N.Y., September 1987. PhD. Thesis (Paul F. Mendler = Nax P. Mendler).
12. Chris Okasaki. Simple and Efficient Purely Functional Queues and Deques. *J. Funct. Program.*, 5(4):583–592, 1995. doi:10.1017/S0956796800001489.
13. Chris Okasaki. Breadth-first numbering: lessons from a small exercise in algorithm design. In Martin Odersky and Philip Wadler, editors, *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming (ICFP '00), Montreal, Canada, September 18-21, 2000.*, ICFP '00, pages 131–136, New York, NY, USA, 2000. ACM. doi:10.1145/351240.351253.

14 John C. Reynolds. Towards a Theory of Type Structure. In B. Robinet, editor, *Programming Symposium*, volume 19 of *Lecture Notes in Computer Science*, pages 408–425. Springer, 1974.

15 John C. Reynolds. Polymorphism is not Set-Theoretic. In Gilles Kahn, David B. MacQueen, and Gordon D. Plotkin, editors, *Semantics of Data Types, International Symposium, Sophia-Antipolis, France, June 27-29, 1984, Proceedings*, volume 173 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 1984. doi:10.1007/3-540-13346-1_7.

A Simpler Undecidability Proof for System F Inhabitation

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

Provability in the intuitionistic second-order propositional logic (resp. inhabitation in the polymorphic lambda-calculus) was shown by Löb to be undecidable in 1976. Since the original proof is heavily condensed, Arts in collaboration with Dekkers provided a fully unfolded argument in 1992 spanning approximately fifty pages. Later in 1997, Urzyczyn developed a different, syntax oriented proof. Each of the above approaches embeds (an undecidable fragment of) first-order predicate logic into second-order propositional logic.

In this work, we develop a simpler undecidability proof by reduction from solvability of Diophantine equations (is there an integer solution to $P(x_1, \ldots, x_n) = 0$ where P is a polynomial with integer coefficients?). Compared to the previous approaches, the given reduction is more accessible for formalization and more comprehensible for didactic purposes. Additionally, we formalize soundness and completeness of the reduction in the Coq proof assistant under the banner of "type theory inside type theory".

2012 ACM Subject Classification Theory of computation → Type theory

Keywords and phrases System F, Lambda Calculus, Inhabitation, Propositional Logic, Provability, Undecidability, Coq, Formalization

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.2

Supplement Material `https://github.com/mrhaandi/ipc2`

Acknowledgements We would like to thank Paweł Urzyczyn for sharing his insights on second order propositional logic provability, which helped to develop the presented results.

1 Introduction

Polymorphic λ-calculus (also known as Girard's system **F** [7] or $\lambda 2$ [2]) is directly related to intuitionistic second-order propositional logic (IPC_2) via the Curry–Howard isomorphism (for an overview see [11]). In particular, provability in the implicational fragment of IPC_2 (is a given formula an IPC_2 theorem?) corresponds to inhabitation in system **F** (given a type, is there a term having that type in system **F**?).

Provability in IPC_2 was shown by Löb to be undecidable [8] (see also [5] for an earlier approach by Gabbay in an extension of IPC_2). Löb's proof is by reduction from provability in first-order predicate logic via a semantic argument. Since the original proof is heavily condensed (14 pages), Arts in collaboration with Dekkers provided a fully unfolded argument [1] (50 pages) reconstructing the original proof. Later, Urzyczyn developed a different, syntax oriented proof showing undecidability of inhabitation in system **F** [13] (6 pages, moderately condensed). Urzyczyn's proof is by reduction from two-counter automata to a fragment of first-order predicate logic to inhabitation in system **F**. In 2010 Sørensen and Urzyczyn [12] gave a general translation of intuitionistic first-order predicate logic, covering the full set of logical connectives, into intuitionistic second-order propositional logic.

In order to show undecidability of provability in IPC_2, each of the above approaches embeds (a fragment of) first-order predicate logic into IPC_2. However, if one is solely interested in a concise and rigorous undecidability proof (e.g. for formalization or didactics), then there is no need to represent an expressive logic.

In this work we provide a reduction from solvability of Diophantine equations (is there an integer solution to $P(x_1, \ldots, x_n) = 0$ where P is a polynomial with integer coefficients?) to inhabitation in system **F**. Compared to the previous approaches, the described reduction is more accessible for formalization and more comprehensible for didactic purposes. Compared to Löb's proof, we separate IPC_2 proof normalization from the main argument. Compared to Urzyczyn's proof, we only need to axiomatize natural number addition and multiplication, instead of a fragment of first-order predicate logic.

Additionally, we formalize [3] soundness and completeness of the reduction in the Coq proof assistant under the banner of "type theory inside type theory".

Organization of the paper. The polymorphic λ-calculus (system **F**) is described in Section 2 together with the associated inhabitation problem (Problem 6). In Section 3 we reduce a decision problem (Problem 9), which is equivalent to solvability of Diophantine equations, to inhabitation in system **F**. Additionally, in Paragraph 3.3 we outline a formalization of soundness (Theorem 27) and completeness (Theorem 19) of the described reduction. We conclude the paper in Section 4.

2 Polymorphic Lambda-Calculus

The Polymorphic Lambda-Calculus (also known as Girard's system **F** [7] or $\lambda 2$ [2]) provides a concise proof notation for the implicational fragment of intuitionistic second-order propositional logic (IPC_2) under the Curry-Howard isomorphism. In this section we assemble necessary prerequisites in order to discuss inhabitation in system **F** (or equivalently provability in IPC_2).

We denote *polymorphic types* (Definition 1) by σ, τ, ρ, where *type variables* are denoted by a, b, c and drawn from the denumerable set $\mathbb{A}$. Conventionally, the operator $\to$ binds more strongly than $\forall$.

▶ **Definition 1** (Polymorphic Types, $\mathbb{T}$). $\mathbb{T} \ni \sigma, \tau, \rho ::= a \mid (\sigma \to \tau) \mid (\forall a.\sigma)$

Type variables that are not *bound* by the operator $\forall$ are *free*, and the set of free type variables in a type σ is denoted by $\mathrm{Var}(\sigma) = \{a \in \mathbb{A} \mid a \text{ is free in } \sigma\}$. A *substitution* of occurrences of a free type variable a in σ by τ is denoted by $\sigma[a := \tau]$.

We denote *Church-style polymorphic λ-terms* (Definition 2) by M, N, where *term variables* are denoted by x, y, z.

▶ **Definition 2** (Church-style Polymorphic λ-Terms).
$M, N ::= x \mid (M\ N) \mid (\lambda x : \sigma.M) \mid (\Lambda a.M) \mid (M\ \tau)$

A *type environment*, denoted by Δ, is a finite set of *type assumptions* having the shape $x : \sigma$ for distinct term variables.

▶ **Definition 3** (Type Environment). $\Delta ::= \{x_1 : \sigma_1, \ldots, x_n : \sigma_n\}$ *where* $x_i \neq x_j$ *for* $i \neq j$

We define the *domain*, the *erasure*, the *extension* of Δ, and the *free type variables* in Δ.

▶ **Definition 4** (Domain, Erasure, Extension, Free Type Variables).

$$\mathrm{dom}(\Delta) = \{x_1, \ldots, x_n\} \qquad |\Delta| = \{\sigma_1, \ldots, \sigma_n\}$$

$$\Delta, x : \sigma = \Delta \cup \{x : \sigma\} \text{ if } x \notin \mathrm{dom}(\Delta) \qquad \mathrm{Var}(\Delta) = \bigcup_{\sigma \in |\Delta|} \mathrm{Var}(\sigma)$$

The rules of the system **F** with *judgements* of shape $\Delta \vdash M : \sigma$ are given below (cf. [11, Section 12]). This system enjoys subject reduction and strong normalization properties.

▶ **Definition 5** (system **F**).

$$\frac{}{\Delta, x : \tau \vdash x : \tau}\ (\mathrm{Ax})$$

$$\frac{\Delta \vdash M : \sigma \to \tau \qquad \Delta \vdash N : \sigma}{\Delta \vdash M\ N : \tau}\ (\to\mathrm{E}) \qquad \frac{\Delta \vdash M : \forall a.\sigma}{\Delta \vdash M\ \tau : \sigma[a := \tau]}\ (\forall\mathrm{E})$$

$$\frac{\Delta, x : \sigma \vdash M : \tau}{\Delta \vdash \lambda x : \sigma.M : \sigma \to \tau}\ (\to\mathrm{I}) \qquad \frac{\Delta \vdash M : \tau \qquad a \notin \mathrm{Var}(\Delta)}{\Delta \vdash \Lambda a.M : \forall a.\tau}\ (\forall\mathrm{I})$$

We sometimes superscript types assigned to subterms in a derivation of a judgement, e.g.

$$\emptyset \vdash \Big(\lambda x : (\forall a.a \to a).\big((x\ (b \to b))^{(b\to b)\to(b\to b)}\ (x\ b)\big)^{b\to b}\Big)\ \Big(\Lambda a.\lambda y : a.y\Big)^{\forall a.a\to a} : b \to b$$

One core decision problem for any typing system is *inhabitation* (Problem 6).

▶ **Problem 6** (Inhabitation, $\Delta \vdash ? : \tau$). *Given a type environment Δ and a type τ, is there a term M such that $\Delta \vdash M : \tau$?*

Inhabitation in system **F** directly corresponds to provability in IPC_2 [11, Section 12] (Proposition 7).

▶ **Proposition 7.** *$\Delta \vdash M : \tau$ iff τ is derivable from $|\Delta|$ in the intuitionistic second-order propositional logic.*

Whenever the particular inhabitant M is immaterial, we write $|\Delta| \vdash \tau$ for $\Delta \vdash M : \tau$. A key property of system **F** is that given a type derivation $\Delta \vdash M : \tau$, there exists a term N^τ in β-normal η-*long* form such that $\Delta \vdash N : \tau$ [13, Lemma 4]. The property of η-longness (Definition 8, cf. *fully applied* in [13]) is defined inductively, taking into account types (ascribed in supersripts) which are assigned to individual subterms.

▶ **Definition 8** (η-longness). *A term M^τ is η-long if one of the following conditions is met*

- *$M^\tau = x^\sigma\ t_1 \ldots t_n$ and $\tau = a$ for some term variable x, type variable a and types or η-long terms $t_1, \ldots, t_n$*
- *$M^\tau = (\lambda x : \sigma.N^\rho)^{\sigma\to\rho}$ where N^ρ is η-long*
- *$M^\tau = (\Lambda a.N^\rho)^{\forall a.\rho}$ where N^ρ is η-long*

We say that N is a *long normal inhabitant* of τ in Δ, if $\Delta \vdash N : \tau$ and N^τ is in β-normal η-long form.

3 Undecidability of Inhabitation

In the remainder of this work we use $\mathbb{N}$ to denote the set of positive integers. As a starting point, we use the following Problem 9, which is undecidable by reduction from solvability of Diophantine equations (for an overview see [9]). In particular, solvability of Diophantine equations in integers is equivalent to solvability of Diophantine equations in $\mathbb{N}$, which by routine subterm decomposition is equivalent to Problem 9.

▶ **Problem 9.** *Given a set* $\mathsf{A} = \{\mathfrak{e}_1, \dots \mathfrak{e}_l\}$ *of constraints over variables* $\mathcal{V} = \{a_1, \dots, a_n\}$ *where each* $\mathfrak{e} \in \mathsf{A}$ *is of shape either* $a \doteq 1$ *or* $a \doteq b + c$ *or* $a \doteq b \cdot c$ *for some* $a, b, c \in \mathcal{V}$*, does there exist a substitution* $\zeta : \mathcal{V} \to \mathbb{N}$ *that satisfies* A*?*

▶ **Proposition 10.** *Problem 9 is undecidable.*

In order to reduce Problem 9 to inhabitation in system **F** it suffices to axiomatize natural number addition and multiplication. Let us fix an instance A of Problem 9 over variables $\mathcal{V} = \{a_1, \dots, a_n\}$. In the remainder of this section we construct the type environment Δ_{A} such that A has a solution iff there exists a term M such that $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$.

For our construction let us fix the type variables $\dagger, u, s, p, \blacktriangle, \bullet_1, \bullet_2, \bullet_3$ and $\bar{i}$ for $i \in \mathbb{N}$. Additionally, for each variable $a_i \in \mathcal{V}$ let us fix the type variable a_i.

Similarly to [13, Section 7], we define the following types to represent particular predicates on natural numbers.

▶ **Definition 11** (Types $\dagger\sigma, U(\sigma), S(\sigma,\tau,\rho), P(\sigma,\tau,\rho)$)**.**

$$\begin{aligned}
\dagger\sigma &= \sigma \to \dagger \\
U(\sigma) &= (\dagger\sigma \to \bullet_1) \to (\sigma \to \bullet_2) \to u \\
S(\sigma,\tau,\rho) &= (\dagger\sigma \to \bullet_1) \to (\dagger\tau \to \bullet_2) \to (\dagger\rho \to \bullet_3) \to s \\
P(\sigma,\tau,\rho) &= (\dagger\sigma \to \bullet_1) \to (\dagger\tau \to \bullet_2) \to (\dagger\rho \to \bullet_3) \to p
\end{aligned}$$

Intuitively, the type $U(\sigma)$ is used to assert that σ represents a natural number, and $S(\sigma,\tau,\rho)$ (resp. $P(\sigma,\tau,\rho)$) is used to assert that the sum (resp. product) of natural numbers represented by σ and τ is represented by ρ. The motivation behind the above encoding (including types $\dagger\sigma$) is of technical nature, leading to convenient inversion lemmas.

Using above types, we represent constraints as follows

▶ **Definition 12** (Constraint Representation)**.**

$$\overline{a \doteq 1} = P(\overline{1}, \overline{1}, a) \qquad \overline{a \doteq b + c} = S(b, c, a) \qquad \overline{a \doteq b \cdot c} = P(b, c, a)$$

Next, we axiomatize finite fragments of natural number arithmetic as follows

▶ **Definition 13** (Type Environments $\Delta_{\mathbb{N}}, \Delta_{\overline{1}}$)**.**

$$\begin{aligned}
\Delta_{\mathbb{N}} = \Big\{ & x_u : \forall a.\Big(U(a) \to \forall b.(U(b) \to S(a,\overline{1},b) \to P(b,\overline{1},b) \to \blacktriangle) \to \blacktriangle\Big), \\
& x_s : \forall abcde.\Big(U(a) \to U(b) \to U(c) \to U(d) \to U(e) \to \\
& \qquad S(a,b,c) \to S(b,\overline{1},d) \to S(c,\overline{1},e) \to (S(a,d,e) \to \blacktriangle) \to \blacktriangle\Big), \\
& x_p : \forall abcde.\Big(U(a) \to U(b) \to U(c) \to U(d) \to U(e) \to \\
& \qquad P(a,b,c) \to S(b,\overline{1},d) \to S(c,a,e) \to (P(a,d,e) \to \blacktriangle) \to \blacktriangle\Big)\Big\} \\
\Delta_{\overline{1}} = \big\{ & y_{U(\overline{1})} : U(\overline{1}), y_{P(\overline{1},\overline{1},\overline{1})} : P(\overline{1},\overline{1},\overline{1})\big\}
\end{aligned}$$

As we will see in the subsequent development, type assumptions in $\Delta_{\mathbb{N}} \cup \Delta_{\overline{1}}$ encompass the following assertions about members of a universe $\mathcal{U}$ which represent natural numbers

- $y_{U(\overline{1})}$ asserts that $\overline{1} \in \mathcal{U}$ and $y_{P(\overline{1},\overline{1},\overline{1})}$ asserts that $\overline{1} \cdot \overline{1} = \overline{1}$
- x_u asserts that for any $a \in \mathcal{U}$ there is $b \in \mathcal{U}$ such that $a + \overline{1} = b$ and $b \cdot \overline{1} = b$
- x_s asserts for $a, b, c, d, e \in \mathcal{U}$: if $a + b = c$, $b + \overline{1} = d$ and $c + \overline{1} = e$, then $a + d = e$
- x_p asserts for $a, b, c, d, e \in \mathcal{U}$: if $a \cdot b = c$, $b + \overline{1} = d$ and $c + a = e$, then $a \cdot d = e$

The choice of $\Delta_{\mathbb{N}}$ is motivated by the fact that a solution of A is supported by an appropriately large finite fragment of natural number arithmetic and does not require the induction principle.

Let the type environment Δ_{A} (Definition 14) encompass the axiomatization of natural number arithmetic together with the assumption that the representation of a solution of A implies $\blacktriangle$. We will reduce solvability of A to $\Delta_{\mathsf{A}} \vdash ? : \blacktriangle$.

▶ **Definition 14** (Type Environments $\Delta_I, \Delta_{\mathsf{A}}$).

$$\Delta_I = \Delta_{\mathbb{N}} \cup \left\{ x_{\mathsf{A}} : \forall a_1 \dots a_n.\big(U(a_1) \to \dots \to U(a_n) \to \overline{\mathfrak{e}_{\overline{1}}} \to \dots \to \overline{\mathfrak{e}_{\overline{\mathsf{I}}}} \to \blacktriangle\big) \right\}$$

$$\Delta_{\mathsf{A}} = \Delta_I \cup \Delta_{\overline{1}}$$

In the above, the type variable $\blacktriangle$ assumes the role of the type variable **false** in [13]. Whereas [13] uses a positive description of first-order predicate logic, we (again, for technical convenience) use doubly-negated conclusions in $\Delta_{\mathbb{N}}$. Following this intuition, the type of x_u corresponds to $\forall a.U(a) \to \neg(\forall b.\neg(U(b) \wedge S(a, \overline{1}, b) \wedge P(b, \overline{1}, b)))$ (cf. list of assertions above). Possibly, we could have used a more natural second-order axiomatization of natural numbers with conventional negation ($\neg\sigma = \sigma \to \forall a.a$) and existential ($\exists a.\sigma = \forall b.((\forall a.(\sigma \to b)) \to b)$) representations. However, both introduce additional universal quantifiers that are neither necessary nor convenient in the proof.

In the remainder of this section we establish completeness (Theorem 19) and soundness (Theorem 27) of the reduction from solvability of A to $\Delta_{\mathsf{A}} \vdash ? : \blacktriangle$.

3.1 Completeness

In this paragraph we show that satisfiability of A implies $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$ for some term M. Intuitively, we derive $|\Delta_{\mathsf{A}}| \vdash \blacktriangle$ in four steps by approaching the goal $\blacktriangle$ many times, each time adding new assumptions. Step 1 introduces representations $\overline{2}, \dots, \overline{\mathsf{N}}$ of natural numbers $2, \dots, \mathsf{N}$, where N is the maximal element in the codomain of some solution of A. Additionally, step 1 introduces assumptions $U(\bar{i})$, $S(\overline{i-1}, \overline{1}, \bar{i})$ and $P(\bar{i}, \overline{1}, \bar{i})$ for $i = 2 \dots \mathsf{N}$. Step 2 introduces information on addition for numbers $1, \dots, \mathsf{N}$, i.e. for $i+j = k \leq \mathsf{N}$ we introduce the assumption $S(\bar{i}, \bar{j}, \bar{k})$. Step 3 introduces information on multiplication for numbers $1, \dots, \mathsf{N}$, i.e. for $i \cdot j = k \leq \mathsf{N}$ we introduce the assumption $P(\bar{i}, \bar{j}, \bar{k})$. Finally, step 4 uses the introduced assumptions to derive $\blacktriangle$ using $x_{\mathsf{A}} : \forall a_1 \dots a_n.\big(U(a_1) \to \dots \to U(a_n) \to \overline{\mathfrak{e}_{\overline{1}}} \to \dots \to \overline{\mathfrak{e}_{\overline{\mathsf{I}}}} \to \blacktriangle\big)$.

For a more accessible presentation of the proof of completeness (Theorem 19), we define type environments $\Delta_U^m, \Delta_S^m, \Delta_P^m$ that contain assumptions for natural numbers up to a bound m that are introduced using x_u. Observe that $\Delta_{\overline{1}} = \Delta_U^1 \cup \Delta_S^1 \cup \Delta_P^1$.

▶ **Definition 15** (Type Environments $\Delta_U^m, \Delta_S^m, \Delta_P^m$). *For $m \in \mathbb{N}$ let*

$$\Delta_U^m = \{ y_{U(\bar{i})} : U(\bar{i}) \mid i = 1 \dots m \}$$

$$\Delta_S^m = \{ y_{S(\overline{i-1},\overline{1},\bar{i})} : S(\overline{i-1}, \overline{1}, \bar{i}) \mid i = 2 \dots m \}$$

$$\Delta_P^m = \{ y_{P(\bar{i},\overline{1},\bar{i})} : P(\bar{i}, \overline{1}, \bar{i}) \mid i = 1 \dots m \}$$

The following Lemmas 16, 17, and 18 each contain the inductive argument used in the outlined steps 1, 2, and 3. Specifically, these lemmas are used to introduce sufficient information on representations of natural numbers to verify a solution of A.

▶ **Lemma 16.** *Let $m \in \mathbb{N}$. If $\Delta_I \cup \Delta_U^{m+1} \cup \Delta_S^{m+1} \cup \Delta_P^{m+1} \vdash N : \blacktriangle$, then $\Delta_I \cup \Delta_U^m \cup \Delta_S^m \cup \Delta_P^m \vdash M : \blacktriangle$ for some M.*

Proof. Immediate using $M = x_u\ \overline{m}\ y_{U(\overline{m})}\ (\Lambda\overline{m+1}.M')$, where
$M' = \lambda y_{U(\overline{m+1})} : U(\overline{m+1}).\lambda y_{S(\overline{m},\overline{1},\overline{m+1})} : S(\overline{m}, \overline{1}, \overline{m+1}).\lambda y_{P(\overline{m+1},\overline{1},\overline{m+1})} : P(\overline{m+1}, \overline{1}, \overline{m+1}).N$. ◀

▶ **Lemma 17.** *Let $i,j,k,m \in \mathbb{N}$ be such that $i,j,k \leq m$ and let $\Delta_S \supseteq \Delta_S^m$ be a type environment such that $(y_{S(\bar{i},\bar{j},\bar{k})} : S(\bar{i},\bar{j},\bar{k})) \in \Delta_S$.*
If $\Delta_I \cup \Delta_U^m \cup \Delta_S \cup \{y_{S(\bar{i},\overline{j+1},\overline{k+1})} : S(\bar{i},\overline{j+1},\overline{k+1})\} \cup \Delta_P^m \vdash N : \blacktriangle$,
then $\Delta_I \cup \Delta_U^m \cup \Delta_S \cup \Delta_P^m \vdash M : \blacktriangle$ for some M.

Proof. Immediate using
$M = x_s\ \bar{i}\ \bar{j}\ \bar{k}\ \overline{j+1}\ \overline{k+1}\ y_{U(\bar{i})}\ y_{U(\bar{j})}\ y_{U(\bar{k})}\ y_{U(\overline{j+1})}\ y_{U(\overline{k+1})}\ y_{S(\bar{i},\bar{j},\bar{k})}\ y_{S(\bar{j},\bar{1},\overline{j+1})}\ y_{S(\bar{k},\bar{1},\overline{k+1})}\ M'$,
where $M' = \lambda y_{S(\bar{i},\overline{j+1},\overline{k+1})} : S(\bar{i},\overline{j+1},\overline{k+1}).N$. ◀

▶ **Lemma 18.** *Let $i,j,k,m \in \mathbb{N}$ be such that $i,j,k \leq m$, $\Delta_S \supseteq \Delta_S^m$ be such that $(y_{S(\bar{k},\bar{i},\overline{k+i})} : S(\bar{k},\bar{i},\overline{k+i})) \in \Delta_S$ and Δ_P be such that $(y_{P(\bar{i},\bar{j},\bar{k})} : P(\bar{i},\bar{j},\bar{k})) \in \Delta_P$.*
If $\Delta_I \cup \Delta_U^m \cup \Delta_S \cup \Delta_P \cup \{y_{P(\bar{i},\overline{j+1},\overline{k+i})} : P(\bar{i},\overline{j+1},\overline{k+i})\} \vdash N : \blacktriangle$,
then $\Delta_I \cup \Delta_U^m \cup \Delta_S \cup \Delta_P \vdash M : \blacktriangle$ for some M.

Proof. Immediate using
$M = x_p\ \bar{i}\ \bar{j}\ \bar{k}\ \overline{j+1}\ \overline{k+i}\ y_{U(\bar{i})}\ y_{U(\bar{j})}\ y_{U(\bar{k})}\ y_{U(\overline{j+1})}\ y_{U(\overline{k+i})}\ y_{P(\bar{i},\bar{j},\bar{k})}\ y_{S(\bar{j},\bar{1},\overline{j+1})}\ y_{S(\bar{k},\bar{i},\overline{k+i})}\ M'$,
where $M' = \lambda y_{P(\bar{i},\overline{j+1},\overline{k+i})} : P(\bar{i},\overline{j+1},\overline{k+i}).N$. ◀

By repeated application of the above Lemmas 16, 17, and 18 we show that a solution of A induces an inhabitant M such that $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$.

▶ **Theorem 19** (Completeness)**.** *If A has a solution, then $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$ for some M.*

Proof. Let $\zeta : \mathcal{V} \to \mathbb{N}$ solve A, and let $\mathsf{N} = \max\{\zeta(a) \mid a \in \mathcal{V}\}$. We derive $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$ in four steps.

Step 1: By repeated application of Lemma 16, in order to derive $|\Delta_{\mathsf{A}}| \vdash \blacktriangle$, it suffices to derive $|\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S^{\mathsf{N}} \cup \Delta_P^{\mathsf{N}}| \vdash \blacktriangle$. Observe that

- For $S(\bar{i},\bar{j},\bar{k}) \in |\Delta_S^{\mathsf{N}}|$ we have $j = 1$ and $i + j = k$
- For $P(\bar{i},\bar{j},\bar{k}) \in |\Delta_P^{\mathsf{N}}|$ we have $j = 1$ and $i \cdot j = k$

Step 2: By repeated application of Lemma 17, in order to derive $|\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S^{\mathsf{N}} \cup \Delta_P^{\mathsf{N}}| \vdash \blacktriangle$, it suffices to derive $|\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S \cup \Delta_P^{\mathsf{N}}| \vdash \blacktriangle$,
where $\Delta_S = \{y_{S(\bar{i},\bar{j},\bar{k})} : S(\bar{i},\bar{j},\bar{k}) \mid i,j,k \in \mathbb{N} \text{ and } i + j = k \leq \mathsf{N}\}$.

Step 3: By repeated application of Lemma 18, in order to derive $|\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S \cup \Delta_P^{\mathsf{N}}| \vdash \blacktriangle$, it suffices to derive $|\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S \cup \Delta_P| \vdash \blacktriangle$,
where $\Delta_P = \{y_{P(\bar{i},\bar{j},\bar{k})} : P(\bar{i},\bar{j},\bar{k}) \mid i,j,k \in \mathbb{N} \text{ and } i \cdot j = k \leq \mathsf{N}\}$.

Step 4: Finally, the claim follows from the following judgement

$$\Delta_I \cup \Delta_U^{\mathsf{N}} \cup \Delta_S \cup \Delta_P \vdash x_{\mathsf{A}}\ \overline{\zeta(a_1)} \ldots \overline{\zeta(a_n)}\ y_{U(\overline{\zeta(a_1)})} \cdots y_{U(\overline{\zeta(a_n)})}\ y_{\overline{\mathfrak{e}_1}} \ldots y_{\overline{\mathfrak{e}_l}} : \blacktriangle$$

In particular, we have

- $\zeta(a_i) \leq \mathsf{N}$ implies $U(\overline{\zeta(a_i)}) \in |\Delta_U^{\mathsf{N}}|$ for $i = 1 \ldots n$
- $\zeta(a) = 1$ implies $\overline{\overline{\zeta(a)} \doteq 1} = P(\bar{1},\bar{1},\bar{1}) \in |\Delta_P|$
- $\zeta(a) = \zeta(b) + \zeta(c) \leq \mathsf{N}$ implies $\overline{\overline{\zeta(a)} \doteq \overline{\zeta(b)} + \overline{\zeta(c)}} = S(\overline{\zeta(b)},\overline{\zeta(c)},\overline{\zeta(a)}) \in |\Delta_S|$
- $\zeta(a) = \zeta(b) \cdot \zeta(c) \leq \mathsf{N}$ implies $\overline{\overline{\zeta(a)} \doteq \overline{\zeta(b)} \cdot \overline{\zeta(c)}} = P(\overline{\zeta(b)},\overline{\zeta(c)},\overline{\zeta(a)}) \in |\Delta_P|$ ◀

3.2 Soundness

In this paragraph we show that $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$ implies satisfiability of A. Intuitively, we show that a derivation of $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$, where M is β-normal and η-long, necessarily completes (parts of) the four steps described in Section 3.1, only adding sound assumptions wrt. addition and multiplication.

Let us define the set of types $\mathcal{C}$ (Definition 20), observing that $\dagger \notin \mathcal{C}$ and $\overline{1} \notin \mathcal{C}$.

▶ **Definition 20** (Set of Types $\mathcal{C}$). $\mathcal{C} = \{u, s, p, \blacktriangle, \bullet_1, \bullet_2, \bullet_3\}$.

We use $\mathcal{C}$, from which any formula in $|\Delta_{\mathsf{A}}|$ is derivable, to hide particular structure of Δ_{A} and identify certain types that are "logically equivalent" wrt. Δ_{A}.

▶ **Lemma 21.** *Let $a, b \in \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ be type variables. If $\mathcal{C} \vdash \dagger a \to \dagger b$, then $a = b$.*

Proof. A long normal inhabitant M of $\dagger a \to \dagger b$ in $\mathcal{C}$ is necessarily of the shape $M = \lambda x : \dagger a.\lambda y : b.(x^{\dagger a}\ y^{b})^{\dagger}$, which implies $a = b$. ◀

▶ **Corollary 22.** *Let σ, τ be types and let $a, b \in \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ be type variables. If $\mathcal{C} \vdash \dagger a \to \dagger\sigma$, $\mathcal{C} \vdash \dagger\sigma \to \dagger\tau$ and $\mathcal{C} \vdash \dagger\tau \to \dagger b$, then $a = b$.*

Using the above Corollary 22 we can lift functions with type variable domain to functions with type domain (Definition 23).

▶ **Definition 23.** *Given a map $[\![\cdot]\!] : \mathcal{U} \to \mathbb{N}$ for some finite set $\mathcal{U} \subseteq \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ of type variables, we define $[\![\cdot]\!]^* : \mathbb{T} \to \mathbb{N}$ by* $[\![\sigma]\!]^* = \begin{cases} [\![a]\!] & \text{if } a \in \mathcal{U},\ \mathcal{C} \vdash \dagger a \to \dagger\sigma \text{ and } \mathcal{C} \vdash \dagger\sigma \to \dagger a \\ \text{undefined} & \text{otherwise, i.e. there is no such } a \end{cases}$

By Corollary 22 the partial map $[\![\cdot]\!]^* : \mathbb{T} \to \mathbb{N}$ is well-defined. Intuitively, the condition $\mathcal{C} \vdash \dagger a \to \dagger\sigma$ and $\mathcal{C} \vdash \dagger\sigma \to \dagger a$ identifies σ with a wrt. Δ_{A} in the sense of the following Lemma 24.

▶ **Lemma 24.** *Let $\sigma \in \mathbb{T}$ be a type and let $\mathcal{U} \subseteq \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ be a finite set of type variables. If $\{s, p, \blacktriangle\} \cup \{U(a) \mid a \in \mathcal{U}\} \vdash U(\sigma)$, then $\mathcal{C} \vdash \dagger a \to \dagger\sigma$ and $\mathcal{C} \vdash \dagger\sigma \to \dagger a$ for some $a \in \mathcal{U}$.*

Proof. A long normal inhabitant M of $U(\sigma)$ is necessarily of the shape

$$M = \lambda x_1 : \dagger\sigma \to \bullet_1.\lambda x_2 : \sigma \to \bullet_2.z^{U(a)}\ (\lambda y_1 : \dagger a.x_1\ N_1^{\dagger\sigma})^{\dagger a \to \bullet_1}\ (\lambda y_2 : a.x_2\ N_2^{\sigma})^{a \to \bullet_2}$$

for some $a \in \mathcal{U}$. Therefore, for $\Gamma = \{s, p, \blacktriangle\} \cup \{U(a) \mid a \in \mathcal{U}\}$ we have

1. $\Gamma, \dagger\sigma \to \bullet_1, \sigma \to \bullet_2, \dagger a \vdash \dagger\sigma$ which implies $\mathcal{C} \vdash \dagger a \to \dagger\sigma$
2. $\Gamma, \dagger\sigma \to \bullet_1, \sigma \to \bullet_2, a \vdash \sigma$ which implies $\mathcal{C} \vdash a \to \sigma$, therefore $\mathcal{C} \vdash \dagger\sigma \to \dagger a$ ◀

▶ **Corollary 25.** *Let $\sigma \in \mathbb{T}$ be a type and let $[\![\cdot]\!] : \mathcal{U} \to \mathbb{N}$ be a map for some finite set $\mathcal{U} \subseteq \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ of type variables. If $\{s, p, \blacktriangle\} \cup \{U(a) \mid a \in \mathcal{U}\} \vdash U(\sigma)$, then $[\![\sigma]\!]^* \in \mathbb{N}$.*

The above Corollary 25 establishes a correspondence between σ and some type variable $a \in \mathcal{U}$ via derivability of $U(\sigma)$. This will allow us to reason about arbitrary (impredicative) instances of types in Δ_{A}. The following Lemma 26 extends this correspondence to sums and products.

▶ **Lemma 26.** *Given a map* $[\![\cdot]\!] : \mathcal{U} \to \mathbb{N}$ *for some finite set* $\mathcal{U} \subseteq \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ *of type variables, let* $\Gamma_S \subseteq \{S(\sigma_1, \sigma_2, \sigma_3) \mid [\![\sigma_1]\!]^* + [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^* \in \mathbb{N}\}$ *and* $\Gamma_P \subseteq \{P(\sigma_1, \sigma_2, \sigma_3) \mid [\![\sigma_1]\!]^* \cdot [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^* \in \mathbb{N}\}$.
For types $\tau_1, \tau_2, \tau_3 \in \mathbb{T}$ *such that* $[\![\tau_1]\!]^*, [\![\tau_2]\!]^*, [\![\tau_3]\!]^* \in \mathbb{N}$ *we have*
(*i*) *If* $\{u, p, \blacktriangle\} \cup \Gamma_S \vdash S(\tau_1, \tau_2, \tau_3)$, *then* $[\![\tau_1]\!]^* + [\![\tau_2]\!]^* = [\![\tau_3]\!]^* \in \mathbb{N}$.
(*ii*) *If* $\{u, s, \blacktriangle\} \cup \Gamma_P \vdash P(\tau_1, \tau_2, \tau_3)$, *then* $[\![\tau_1]\!]^* \cdot [\![\tau_2]\!]^* = [\![\tau_3]\!]^* \in \mathbb{N}$.

Proof. For (*i*), let $\Gamma = \{u, p, \blacktriangle\} \cup \Gamma_S$ and assume $\Gamma \vdash S(\tau_1, \tau_2, \tau_3)$. A long normal inhabitant M of $S(\tau_1, \tau_2, \tau_3)$ is necessarily of the shape

$$M = \lambda x_1 : \dagger\tau_1 \to \bullet_1 . \lambda x_2 : \dagger\tau_2 \to \bullet_2 . \lambda x_3 : \dagger\tau_3 \to \bullet_3 . z^{S(\sigma_1,\sigma_2,\sigma_3)}\ N_1^{\dagger\sigma_1 \to \bullet_1}\ N_2^{\dagger\sigma_2 \to \bullet_2}\ N_3^{\dagger\sigma_3 \to \bullet_3}$$

where $N_i = (\lambda y_i : \dagger\sigma_i . x_i\ L_i^{\dagger\tau_i})$ for $i = 1, 2, 3$ and $S(\sigma_1, \sigma_2, \sigma_3) \in \Gamma_S$.

Therefore, we have $\Gamma, \dagger\tau_1 \to \bullet_1, \dagger\tau_2 \to \bullet_2, \dagger\tau_3 \to \bullet_3 \vdash \dagger\sigma_i \to \dagger\tau_i$ for $i = 1, 2, 3$, which implies $\mathcal{C} \vdash \dagger\sigma_i \to \dagger\tau_i$ for $i = 1, 2, 3$. Additionally, by Definition 23 there exist type variables $a_1, a_2, a_3, b_1, b_2, b_3 \in \mathcal{U}$ such that $\mathcal{C} \vdash \dagger a_i \to \dagger\sigma_i$ and $\mathcal{C} \vdash \dagger\tau_i \to \dagger b_i$ for $i = 1, 2, 3$. By Corollary 22, we obtain $[\![\sigma_i]\!]^* = [\![\tau_i]\!]^*$ for $i = 1, 2, 3$, which implies the claim.

The proof of (*ii*) is analogous to the proof of (*i*). ◀

Finally, we establish soundness of our reduction in the following Theorem 27.

▶ **Theorem 27** (Soundness). *If* $\Delta_{\mathsf{A}} \vdash M : \blacktriangle$ *for some* M, *then* A *has a solution.*

Proof. We show a more general claim. Given a map $[\![\cdot]\!] : \mathcal{U} \to \mathbb{N}$ for some finite set $\mathcal{U} \subseteq \mathbb{A} \setminus (\mathcal{C} \cup \{\dagger\})$ of type variables such that $\overline{1} \in \mathcal{U}$ and $[\![\overline{1}]\!] = 1$, let $\Delta = \Delta_I \cup \Delta_U \cup \Delta_S \cup \Delta_P$ such that

$$|\Delta_U| = \{U(a) \mid a \in \mathcal{U}\}$$
$$|\Delta_S| \subseteq \{S(\sigma_1, \sigma_2, \sigma_3) \mid [\![\sigma_1]\!]^* + [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^* \in \mathbb{N}\}$$
$$|\Delta_P| \subseteq \{P(\sigma_1, \sigma_2, \sigma_3) \mid [\![\sigma_1]\!]^* \cdot [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^* \in \mathbb{N}\}$$

We show that $|\Delta| \vdash \blacktriangle$ implies that A has a solution.

Assume $|\Delta| \vdash \blacktriangle$, then there exists a long normal form M such that $\Delta \vdash M : \blacktriangle$. We proceed by induction on the depth of M, which necessarily has one of the following shapes:

- $x_u \sigma N^{U(\sigma)} (\Lambda b . \lambda y_u : U(b) . \lambda y_s : S(\sigma, \overline{1}, b) . \lambda y_p : P(b, \overline{1}, b) . M_1^{\blacktriangle})$:
  Wlog. b, y_u, y_s, y_p are fresh. We have
  - $\Delta \vdash N : U(\sigma)$, therefore $[\![\sigma]\!]^* \in \mathbb{N}$ by Corollary 25.
  - $\Delta, y_u : U(b), y_s : S(\sigma, \overline{1}, b), y_p : P(b, \overline{1}, b) \vdash M_1 : \blacktriangle$.

  For $\mathcal{U}' = \mathcal{U} \cup \{b\}$ extending the domain of $[\![\cdot]\!]$ to b by $[\![b]\!] := [\![\sigma]\!]^* + 1$, $\Delta'_U = \Delta_U \cup \{y_u : U(b)\}$, $\Delta'_S = \Delta_S \cup \{y_s : S(\sigma, \overline{1}, b)\}$ and $\Delta'_P := \Delta_P \cup \{y_p : P(b, \overline{1}, b)\}$, we have that $\Delta_I \cup \Delta'_U \cup \Delta'_S \cup \Delta'_P \vdash M_1 : \blacktriangle$. Since $[\![b]\!]^* = [\![b]\!] = [\![\sigma]\!]^* + 1 = [\![\sigma]\!]^* + [\![\overline{1}]\!]$ and $[\![b]\!]^* = [\![b]\!]^* \cdot [\![\overline{1}]\!]^*$, by the induction hypothesis we obtain the claim.
- $x_s \sigma_1 \dots \sigma_5 N_1^{U(\sigma_1)} \dots N_5^{U(\sigma_5)} L_1^{S(\sigma_1,\sigma_2,\sigma_3)} L_2^{S(\sigma_2,\overline{1},\sigma_4)} L_3^{S(\sigma_3,\overline{1},\sigma_5)} (\lambda y_s : S(\sigma_1, \sigma_4, \sigma_5) . M_1^{\blacktriangle})$:
  Wlog. y_s is fresh. We have
  - $\Delta \vdash N_i : U(\sigma_i)$, therefore $[\![\sigma_i]\!]^* \in \mathbb{N}$ for $i = 1 \dots 5$ by Corollary 25.
  - $\Delta \vdash L_1 : S(\sigma_1, \sigma_2, \sigma_3)$, $\Delta \vdash L_2 : S(\sigma_2, \overline{1}, \sigma_4)$ and $\Delta \vdash L_3 : S(\sigma_3, \overline{1}, \sigma_5)$. Therefore, $[\![\sigma_1]\!]^* + [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^*$, $[\![\sigma_2]\!]^* + [\![\overline{1}]\!]^* = [\![\sigma_4]\!]^*$ and $[\![\sigma_3]\!]^* + [\![\overline{1}]\!]^* = [\![\sigma_5]\!]^*$ by Lemma 26.
  - $\Delta, y_s : S(\sigma_1, \sigma_4, \sigma_5) \vdash M_1 : \blacktriangle$

  For $\Delta'_S = \Delta_S \cup \{y_s : S(\sigma_1, \sigma_4, \sigma_5)\}$ we have $\Delta_I \cup \Delta_U \cup \Delta'_S \cup \Delta_P \vdash M_1 : \blacktriangle$. Since $[\![\sigma_5]\!]^* = [\![\sigma_3]\!]^* + [\![\overline{1}]\!]^* = [\![\sigma_1]\!]^* + [\![\sigma_2]\!]^* + [\![\overline{1}]\!]^* = [\![\sigma_1]\!]^* + [\![\sigma_4]\!]^*$, by the induction hypothesis we obtain the claim.

- $x_p\sigma_1 \dots \sigma_5 N_1^{U(\sigma_1)} \dots N_5^{U(\sigma_5)} L_1^{P(\sigma_1,\sigma_2,\sigma_3)} L_2^{S(\sigma_2,\overline{1},\sigma_4)} L_3^{S(\sigma_3,\sigma_1,\sigma_5)} (\lambda y_p : P(\sigma_1,\sigma_4,\sigma_5).M_1^{\blacktriangle})$:
  Wlog. y_p is fresh. We have
  - $\Delta \vdash N_i : U(\sigma_i)$, therefore $[\![\sigma_i]\!]^* \in \mathbb{N}$ for $i = 1 \dots 5$ by Corollary 25.
  - $\Delta \vdash L_1 : P(\sigma_1,\sigma_2,\sigma_3)$, $\Delta \vdash L_2 : S(\sigma_2,\overline{1},\sigma_4)$ and $\Delta \vdash L_3 : S(\sigma_3,\sigma_1,\sigma_5)$. Therefore, $[\![\sigma_1]\!]^* \cdot [\![\sigma_2]\!]^* = [\![\sigma_3]\!]^*$, $[\![\sigma_2]\!]^* + [\![\overline{1}]\!]^* = [\![\sigma_4]\!]^*$ and $[\![\sigma_3]\!]^* + [\![\sigma_1]\!]^* = [\![\sigma_5]\!]^*$ by Lemma 26.
  - $\Delta, y_p : P(\sigma_1,\sigma_4,\sigma_5) \vdash M_1 : \blacktriangle$

  For $\Delta'_P = \Delta_P \cup \{y_p : P(\sigma_1,\sigma_4,\sigma_5)\}$ we have $\Delta_I \cup \Delta_U \cup \Delta_S \cup \Delta'_P \vdash M_1 : \blacktriangle$. Since $[\![\sigma_5]\!]^* = [\![\sigma_3]\!]^* + [\![\sigma_1]\!]^* = [\![\sigma_1]\!]^* \cdot [\![\sigma_2]\!]^* + [\![\sigma_1]\!]^* = [\![\sigma_1]\!]^* \cdot ([\![\sigma_2]\!]^* + [\![\overline{1}]\!]^*) = [\![\sigma_1]\!]^* \cdot [\![\sigma_4]\!]^*$, by the induction hypothesis we obtain the claim.
- $x_{\mathsf{A}}\sigma_1 \dots \sigma_n N_1^{U(\sigma_1)} \dots N_n^{U(\sigma_n)} L_1^{\overline{\mathfrak{e}}_1[a_i:=\sigma_i \mid i=1\dots n]} \dots L_l^{\overline{\mathfrak{e}}_l[a_i:=\sigma_i \mid i=1\dots n]}$:
  We have $\Delta \vdash N_i : U(\sigma_i)$, therefore $[\![\sigma_i]\!]^* \in \mathbb{N}$ for $i = 1 \dots n$ by Corollary 25. We show that the map $a_i \mapsto [\![\sigma_i]\!]^*$ satisfies each $\mathfrak{e}_j \in \mathsf{A}$ by distinguishing the following cases for $\mathfrak{e}_j$:
  Case $a_i \doteq 1$: We have $\overline{\mathfrak{e}_j}[a_i := \sigma_i \mid 1 = 1 \dots n] = P(\overline{1},\overline{1},\sigma_i)$. Since $\Delta \vdash L_j : P(\overline{1},\overline{1},\sigma_i)$, we have $[\![\sigma_i]\!]^* = [\![\overline{1}]\!]^* \cdot [\![\overline{1}]\!]^* = 1$ by Lemma 26.
  Case $a_{i_1} \doteq a_{i_2} + a_{i_3}$: We have $\overline{\mathfrak{e}_j}[a_i := \sigma_i \mid 1 = 1 \dots n] = S(\sigma_{i_2},\sigma_{i_3},\sigma_{i_1})$. Since $\Delta \vdash L_j : S(\sigma_{i_2},\sigma_{i_3},\sigma_{i_1})$, we have $[\![\sigma_{i_1}]\!]^* = [\![\sigma_{i_2}]\!]^* + [\![\sigma_{i_3}]\!]^*$ by Lemma 26.
  Case $a_{i_1} \doteq a_{i_2} \cdot a_{i_3}$: We have $\overline{\mathfrak{e}_j}[a_i := \sigma_i \mid 1 = 1 \dots n] = P(\sigma_{i_2},\sigma_{i_3},\sigma_{i_1})$. Since $\Delta \vdash L_j : P(\sigma_{i_2},\sigma_{i_3},\sigma_{i_1})$, we have $[\![\sigma_{i_1}]\!]^* = [\![\sigma_{i_2}]\!]^* \cdot [\![\sigma_{i_3}]\!]^*$ by Lemma 26. ◀

3.3 Formalization

In this paragraph we outline a formalization [3] of the above soundness (Theorem 27) and completeness (Theorem 19) results in Coq 8.8 using the SSReflect proof methodology. The formalization spans 4000 lines of code, of which three quarters is boilerplate.

The main result is formalized in `MainResult.v` as

```
Theorem correctness : ∀ (ds : list diophantine), Diophantine.solvable ds ↔
  derivation (ΓI ds ++ [U one; P one one one]) triangle.
```

In the above, constraints of shape either $a \doteq 1$ or $a \doteq b + c$ or $a \doteq b \cdot c$ that are used in Problem 9 are captured in `Diophantine.v` by the inductive type `Inductive diophantine : Set`. Derivability in system **F** (or rather IPC_2) is formalized in `Derivations.v` by the inductive type

```
Inductive derivation (Γ : list formula) : formula → Prop
```

The property of long normal inhabitation (reflecting Definition 8) is internalized in the definition of inductive type (also containing a bound on the depth of the derivation as the first parameter)

```
Inductive normal_derivation : nat → list formula → formula → Prop
```

For an in-depth analysis of type derivations in system **F** see [6]. Normalization of system **F** and existence of η-long inhabitants, i.e. completeness of `normal_derivation` wrt. `derivation` is (at the time of writing) not part of the formalization

```
Axiom normal_derivation_completeness : ∀ (Γ : list formula) (s: formula),
  derivation Γ s → ∃ (n : nat), normal_derivation n Γ s.
```

whereas soundness of `normal_derivation` wrt. `derivation` is shown by

```
Theorem normal_derivation_soundness : ∀ (n : nat) (Γ : list formula) (s: formula),
  normal_derivation n Γ s → derivation Γs.
```

The more general claim that is used in the proof of soundness (Theorem 27) is formalized in `Soundness.v` as

```
Theorem soundness : ∀ (n : nat) (ΓU ΓS ΓP : list formula),
  (∀ {s : formula}, In s ΓU → represents_nat s) →
  (∀ {s : formula}, In s ΓS → encodes_sum s) →
  (∀ {s : formula}, In s ΓP → encodes_prod s) →
  ∀ (ds : list diophantine),
  normal_derivation n ((Encoding.ΓI ds) ++ ΓU ++ ΓS ++ ΓP) Encoding.triangle →
  Diophantine.solvable ds.
```

Completeness (Theorem 19) is formalized in `Completeness.v` as

```
Lemma completeness : ∀ (ds : list diophantine), Diophantine.solvable ds →
  derivation (ΓI ds ++ [U one; P one one one]) triangle.
```

where the first three steps in the proof of Theorem 19 are formalized individually as `Theorem completeness_U`, `Theorem completeness_S`, and `Theorem completeness_P`.

At the time of writing, theorems `soundness` and `completeness` use only the above axiom `normal_derivation_completeness` as an assumption that is not formally proven.

Several aspects of the "informal" proof, at first glance, appear problematic and are clarified in the formal proof. In Definition 23 we partially define an interpretation $[\![\cdot]\!]^*$ of arbitrary types as natural numbers based on derivability in system **F**. Not only is derivability undecidable, but it is the actual subject of our analysis. The map $[\![\cdot]\!]^*$ is formalized in `Encoding.v` as

```
Inductive interpretation (s : formula) (n : nat) : Prop
```

and its well-definedness is shown in `Soundness.v` by

```
Lemma interpretation_soundness : ∀ (s : formula) (m1 m2 : nat),
  interpretation s m1 → interpretation s m2 → m1 = m2.
```

The absence of classical principles or the axiom of choice (resp. Hilbert's epsilon) as assumptions in our main result ensures that the whole argument is constructive.

Another aspect elaborated in the formal proof is the argumentation based on the necessary shape of long normal inhabitants. Clearly, a complete case analysis of all imaginable inhabitants would clutter an "informal" proof, that is supposed to focus on interesting cases. Luckily, the formal proof can utilize numerous tactics to deal with the trivial cases automatically. Most prominently, the tactic `decompose_USP` implemented in `Soundness.v` discovers and transforms suitable assumptions by full case analysis to apply Lemma 26.

4 Conclusion

This work contains the (as of yet) simplest, syntax oriented proof that inhabitation in system **F** (resp. provability in intuitionistic second-order propositional logic) is undecidable. The proof is by reduction from (a variant of) solvability of Diophantine equations. In spirit, the reduction can be considered an instance of Sørensen's and Urzyczyn's reduction from provability in first-order predicate logic to provability in second-order propositional logic. Additionally, we formalized soundness and completeness results in the Coq proof assistant.

The next step is to eliminate the axiom regarding existence of long normal inhabitants in system **F** by using existing work [10]. In near future, we envision to embed the formalization into the larger framework of computational reductions in Coq [4] already containing a collection of formalized reductions that are used in undecidability results.

References

1 T. Arts and W. Dekkers. Embedding first order predicate logic in second order propositional logic. *Technical report 93-02, Katholieke Universiteit Nijmegen*, 1993.

2 H. Barendregt. Introduction to Generalized Type Systems. *J. Funct. Program.*, 1(2):125–154, 1991.

3 A. Dudenhefner. Reduction from Diophantine equations to provability in IPC2 / System F. `https://github.com/mrhaandi/ipc2`. Accessed: 2018-09-18.

4 Y. Forster, E. Heiter, and G. Smolka. Verification of PCP-Related Computational Reductions in Coq. In *Interactive Theorem Proving - 9th International Conference, ITP 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, pages 253–269, 2018. `doi:10.1007/978-3-319-94821-8_15`.

5 D. M. Gabbay. On 2nd order intuitionistic propositional calculus with full comprehension. *Archive for Mathematical Logic*, 16(3):177–186, 1974.

6 P. Giannini and S. Ronchi Della Rocca. Characterization of typings in polymorphic type discipline. In *Proceedings of the Third Annual Symposium on Logic in Computer Science (LICS '88), Edinburgh, Scotland, UK, July 5-8, 1988*, pages 61–70, 1988. `doi:10.1109/LICS.1988.5101`.

7 J. Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII, 1972.

8 M. H. Löb. Embedding First Order Predicate Logic in Fragments of Intuitionistic Logic. *J. Symb. Log.*, 41(4):705–718, 1976. `doi:10.2307/2272390`.

9 D. Martin. Hilbert's tenth problem is unsolvable. *The American Mathematical Monthly*, 80(3):233–269, 1973.

10 K. Sakaguchi. A Formalization of Typed and Untyped lambda-Calculi in SSReflect-Coq and Agda2. `https://github.com/pi8027/lambda-calculus`. Accessed: 2019-04-02.

11 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.

12 M. H. Sørensen and P. Urzyczyn. A Syntactic Embedding of Predicate Logic into Second-Order Propositional Logic. *Notre Dame Journal of Formal Logic*, 51(4):457–473, 2010. `doi:10.1215/00294527-2010-029`.

13 P. Urzyczyn. Inhabitation in Typed Lambda-Calculi (A Syntactic Approach). In *Typed Lambda Calculi and Applications, Third International Conference on Typed Lambda Calculi and Applications, TLCA '97, Nancy, France, April 2-4, 1997, Proceedings*, pages 373–389, 1997. `doi:10.1007/3-540-62688-3_47`.

Dependent Sums and Dependent Products in Bishop's Set Theory

Iosif Petrakis
Ludwig-Maximilians-Universität Munich, Theresienstrasse 39, Germany
http://www.mathematik.uni-muenchen.de/~petrakis/
petrakis@math.lmu.de

Abstract

According to the standard, non type-theoretic accounts of Bishop's constructivism (BISH), dependent functions are not necessary to BISH. Dependent functions though, are explicitly used by Bishop in his definition of the intersection of a family of subsets, and they are necessary to the definition of arbitrary products. In this paper we present the basic notions and principles of CSFT, a semi-formal constructive theory of sets and functions intended to be a minimal, adequate and faithful, in Feferman's sense, semi-formalisation of Bishop's set theory (BST). We define the notions of dependent sum (or exterior union) and dependent product of set-indexed families of sets within CSFT, and we prove the distributivity of $\prod$ over $\sum$ i.e., the translation of the type-theoretic axiom of choice within CSFT. We also define the notions of dependent sum (or interior union) and dependent product of set-indexed families of subsets within CSFT. For these definitions we extend BST with the universe of sets $\mathbb{V}_0$ and the universe of functions $\mathbb{V}_1$.

2012 ACM Subject Classification Theory of computation → Constructive mathematics

Keywords and phrases Bishop's constructive mathematics, Martin-Löf's type theory, dependent sums, dependent products, type-theoretic axiom of choice

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.3

Acknowledgements I want to thank the Hausdorff Research Institute for Mathematics (HIM) for providing me with ideal conditions of work as a temporary project fellow of the trimester program "Types, Sets and Constructions" (July-August 2018). It was during this stay of mine in Bonn that most of the research around this paper was carried out.

1 Introduction

Bishop's original approach to constructive mathematics, developed in his seminal book *Foundations of Constructive Analysis*, was an important motivation to Martin-Löf's type theory (MLTT). Martin-Löf opened his first published paper on type theory ([23], p. 73) as follows.

> The theory of types with which we shall be concerned is intended to be a full scale system for formalizing intuitionistic mathematics as developed, for example, in the book of Bishop.

As Martin-Löf explains in [22], p. 13, he got access to Bishop's book only shortly after his own book on constructive mathematics [22] was finished. A surprising historical fact is that the first who considered a type-theoretic system as a formal system for Bishop's book [5] was Bishop himself. In the unpublished manuscript [6] Bishop developed an extensional dependent type theory with one universe as a formal system for his book. In the also unpublished manuscript [7] Bishop elaborated the implementation of his type theory into Algol. A similar pattern is followed in [8], where, influenced by Gödel's Dialectica interpretation, Bishop introduced Σ, a variant of HA^{ω}, as a formal system for his book, and discussed the implementation of Σ into Algol (see [8], p. 70).

The question $\boldsymbol{Q}$ of finding a formal system suitable for Bishop's system of informal constructive mathematics BISH was a major question in the foundational studies of the 1970's. Myhill's system CST, introduced in [26], and later Aczel's CZF (see [1]), Friedman's system B, developed in [18], and Feferman's system of explicit mathematics T_0 (see [16] and [17]), are some of the systems motivated by $\boldsymbol{Q}$, but soon developed independently from it. These systems were influenced a lot from the classical Zermelo-Fraenkel set theory, and could be described as "top-down" approaches to $\boldsymbol{Q}$, as they have many "unexpected" features with respect to BISH[1]. Beeson's systems S and S_0 in [3], and Greenleaf's system of liberal constructive set theory LCST in [19] were dedicated to $\boldsymbol{Q}$. Especially Beeson tried to find a faithful and adequate formalisation of BISH, and by including a serious amount of proof relevance to his systems stands in between the set-theoretic, proof-irrelevant point of view and the type-theoretic, proof-relevant point of view.

All aforementioned systems though, were not really "tested" with respect to BISH. Only very small parts of BISH were actually implemented in them, and their adequacy for BISH was mainly a claim, rather than a shown fact. The implementation of Bishop's constructivism within a formal system for it was taken seriously in the type-theoretic formalisations of BISH, and especially in the work of Coquand (see e.g., [12] and [13]), Palmgren (see e.g., [20] and the collaborative work [11]), the Nuprl research group of Constable (see e.g., [15]), and the Minimalist Foundation of Sambin and Maietti (see [37] and [21]).

Bishop's (informal) set theory (BST), developed in Chapter 3 of [5] (or [9]), is reflected in MLTT through the theory of setoids (see especially the work of Palmgren [27]-[29]). The identity type of MLTT (see [24]) though, has no counterpart in BST, a fact with many consequences, as e.g., the existence in MLTT of a free setoid from a given type (see [28], p. 90), a result crucial to the proof of the presentation axiom in MLTT (see [11], p. 75).

The standard, non type-theoretic view regarding dependency within BST is that dependent functions are not necessary. Dependent functions though, do appear explicitly in Bishop's definition of the intersection of $\bigcap_{t\in T}\lambda(t)$, where T is an inhabited set and λ is a family of subsets of some set X indexed by T. In [5], p. 65, and in [9], p. 70, Bishop writes that "..., an element u of $\bigcap_{t\in T}\lambda(t)$ is a rule that associates an element a_t of $\lambda(t)$ to each element t of T". Dependent functions are also necessary to the definition of products of families of sets indexed by an arbitrary set, and can be avoided, if one is restricted to countable products only. Although Bishop himself considered e.g., only countable products of metric spaces, the constructive development of general algebra (see [25]), or general topology (see e.g., [30], [32], [31], and [33]), require the use of arbitrary products, hence the use of dependent functions. As we noted above, Bishop also defined in [6] a notion of dependent types within his type-theoretic system for BISH.

Currently, we revisit question $\boldsymbol{Q}$ in [34] and [35], aiming at a minimal, adequate and faithful formalisation of BST. For that we elaborate a semi-formal[2], constructive set and function theory (CSFT), as the first necessary step to an adequate and faithful formalisation of BST. Although a universe of sets $\mathbb{V}_0$ and a universe of functions $\mathbb{V}_1$ are included in CSFT,

[1] Using Feferman's terminology from [17], these formal systems are not, in our view, *faithful* to BISH, as they contain concepts or axioms that do not appear, neither explicitly nor implicitly, in BISH. Feferman also introduced the notion of an adequate formalisation T of a body of informal mathematics M. Namely, T is *adequate* for M, if every concept, argument, and result of M is represented by a concept, proof, and a theorem, respectively, of T (see also [3], p. 153).

[2] The theory CSFT is semi-formal in the following sense: it fills the "gaps" in Bishop's presentation of the fundamental concepts of BST in a minimal way such that an adequate and faithful formalisation of CSFT can be generated. Hence, CSFT stands between Bishop's informal and incomplete theory of sets and a suitable formal system for it.

and not explicitly mentioned in BST, in section 5 we explain why these classes are implicit[3] in BST. The somewhat "silent" existence of dependency in BISH is replaced by a central presence in CSFT. This is necessary, if we want to make some very basic definitions in BISH precise enough to be formalised.

2 Basic notions of CSFT

Next we briefly present those fundamentals of CSFT required to the material presented in the following sections. A complete presentation is planned to be included in [35].

The general logical framework of (a formalisation of) CSFT is a kind of a many-sorted intuitionistic first-order predicate logic with equality ($:=$). The expression[4] $a := b$ is to be read as "a is by definition equal to b". Similarly, the expression $P :\Leftrightarrow Q$ is read as "P is by definition equivalent to Q". The basic primitives of CSFT are the set of natural numbers $\mathbb{N}$, equipped with its basic equality $=_{\mathbb{N}}$, operations and order, a primitive notion of n-tuple of given objects, for every natural n larger than 2, an undefined notion of *finite routine*, or *construction*, or *algorithm*, and the *assignment routines* $\mathbf{pr}_i(a_1, \ldots, a_n) := a_i$, for every i between 1 and n, and for every n larger than 2, where an assignment routine is defined as a certain finite routine.

A *defined totality* X is defined by a membership condition $\mathcal{M}_X$ i.e., $x \in X :\Leftrightarrow \mathcal{M}_X(x)$, and $\mathcal{M}_X(x)$ is the *membership formula* for X. If X, Y are defined totalities with membership formulas $\mathcal{M}_X$ and $\mathcal{M}_Y$, respectively, we say that X and Y are *definitionally equal*, $X := Y$, if $\big[\mathcal{M}_X(x) :\Leftrightarrow \mathcal{M}_Y(x)\big]$. A *totality* is either the primitive $\mathbb{N}$ or a defined totality. A totality X is called *inhabited*, if there is $x_0 \in X$. A *defined totality with equality* is a defined totality X equipped with an equality condition $\mathcal{E}_X$ i.e., $x =_X y :\Leftrightarrow \mathcal{E}_X(x, y)$, where the *equality formula* $\mathcal{E}_X(x, y)$ satisfies the defining conditions of an equivalence relation. A *defined set* is a defined totality with equality such that the membership formula $\mathcal{M}_X(x)$ for X represents a construction, or a finite routine. A *set* is the primitive $\mathbb{N}$ or a defined set. If $\mathcal{M}_X(x)$ does not reflect a construction, then X is a *class*. E.g., if X, Y are sets, their *product* $X \times Y$ is the defined totality with equality given by

$$z \in X \times Y :\Leftrightarrow \exists_{x \in X} \exists_{y \in Y} \big(z := (x, y)\big),$$

$$z =_{X \times Y} w :\Leftrightarrow \mathbf{pr}_1(z) =_X \mathbf{pr}_1(w) \;\&\; \mathbf{pr}_2(z) =_Y \mathbf{pr}_2(w).$$

For simplicity, we usually write an equality formula, as that for $X \times Y$, as follows: $(x, y) =_{X \times Y} (x', y') :\Leftrightarrow x =_X x' \;\&\; y =_Y y'$. In contrast to MLTT, we allow the use of the equality $:=$ within membership formulas (only). Clearly, if X, Y are sets, then $X \times Y$ is also a set, since the construction of an element of $X \times Y$ is reduced to the construction of an element of X and of an element of Y.

If X, Y are totalities, an *assignment routine* $f : X \rightsquigarrow Y$ from X to Y is a finite routine assigning an element y of Y i.e., $\mathcal{M}_Y(y)$), to each given element x of X i.e., $\mathcal{M}_X(x)$. In this case we write $f(x) := y$. E.g., the assignment routine $\mathtt{pr}_X$ from $X \times Y$ to X is defined by $\mathtt{pr}_X(x, y) := \mathbf{pr}_1(x, y) := x$, for every $(x, y) \in X \times Y$. If X, Y, Z are totalities, $f : X \rightsquigarrow Y$ and $g : Y \rightsquigarrow Z$ are assignment routines, the *composition* assignment routine $g \circ f : X \rightsquigarrow Z$ is defined by $(g \circ f)(x) := g(f(x))$, for every $x \in X$. If f and g are assignment routines from X to Y, they are *definitionally equal*, $f := g$, if $\forall_{x \in X}\big(f(x) := g(x)\big)$. E.g., for the

[3] In [34] we do not use $\mathbb{V}_1$, but instead we consider a dependent assignment routine as a primitive notion.
[4] Bishop's notation for definitional equality is $a \equiv b$.

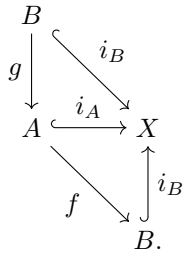
assignment routine $\mathrm{id}_X : X \rightsquigarrow X$, defined by $\mathrm{id}_X(x) := x$, for every $x \in X$, we have that $f \circ \mathrm{id}_X := f$. If X, Y are sets, we call an assignment routine from X to Y an *operation*, while a *function* $f : X \to Y$ from a set X to a set Y is an extensional operation from X to Y i.e., $f(x) =_Y f(x')$, for every $x, x' \in X$ such that $x =_X x'$. A function $f : X \to Y$ is an *embedding* of X into Y, if $x =_X x'$, whenever $f(x) =_Y f(x')$. We denote such an embedding by $f : X \hookrightarrow Y$. If X, Y are sets, the defined totality with equality $\mathbb{F}(X, Y)$ of functions from X to Y, defined by

$$z \in \mathbb{F}(X, Y) :\Leftrightarrow z := f : X \to Y,$$

$$f =_{\mathbb{F}(X,Y)} g :\Leftrightarrow \forall_{x \in X}\big(f(x) =_Y f(y)\big),$$

is a set, as $\mathcal{M}_{\mathbb{F}(X,Y)}(z)$ represents a construction. A *subset* of a set X is a pair (A, i_A), where A is a set and $i_A : A \hookrightarrow X$. The *powerset* of X is the defined totality $\mathcal{P}(X)$ of subsets of X with equality defined by

$$(A, i_A) =_{\mathcal{P}(X)} (B, i_B) :\Leftrightarrow \exists_{f:A\to B}\exists_{g:B\to A}\big(i_A \circ g =_{\mathbb{F}(B,X)} i_B \;\&\; i_B \circ f =_{\mathbb{F}(A,X)} i_A\big)$$



If f and g realize the equality between (A, i_A) and (B, i_B) in $\mathcal{P}(X)$, we write $(f, g) : (A, i_A) =_{\mathcal{P}(X)} (B, i_B)$. For simplicity, we may write $A =_{\mathcal{P}(X)} B$ instead of $(A, i_A) =_{\mathcal{P}(X)} (B, i_B)$. To construct an element of $\mathcal{P}(X)$ one needs to construct a set A and an embedding from A to X. This membership condition does not express a construction that can be carried out in a finite time, since there is no known finite algorithm to construct a set. Consequently, $\mathcal{P}(X)$ is a class. If $P(x)$ is an *extensional property* on X i.e., a formula satisfying $\forall_{x,y\in X}\big(x =_X y \;\&\; P(x) \Rightarrow P(y)\big)$, the totality with equality X_P is defined by

$$x \in X_P :\Leftrightarrow x \in X \;\&\; P(x),$$

and $x =_{X_P} x' :\Leftrightarrow x =_X x'$. We may also use the notation $\{x \in X \mid P(x)\}$ for X_P. If X is a set, then X_P is a set, and the pair (X_P, i_{X_P}), where $i_{X_P} : X_P \hookrightarrow X$ is defined by $i_{X_P}(x) := x$, for every $x \in X_P$, is in $\mathcal{P}(X)$. We call X_P the *extensional subset* of X generated by $P(x)$. If X is a set, the *diagonal* of X is the set

$$D(X) := \{(x, y) \in X \times X \mid x =_X y\}$$

i.e., the extensional subset of $X \times X$ generated by $P(x, y) :\Leftrightarrow x =_X y$ on $X \times X$.

If (A, i_A) and (B, i_B) are subsets of X, their *intersection* $A \cap B$ is defined by

$$A \cap B := \{(a, b) \in A \times B \mid i_A(a) =_X i_B(b)\}.$$

Let $i : A \cap B \rightsquigarrow X$ the assignment routine defined by $i(a, b) := i_A(\mathbf{pr}_1(a, b)) := i_A(a)$, for every $(a, b) \in A \cap B$. The equality on $A \cap B$ is defined by $(a, b) =_{A\cap B} (a', b') :\Leftrightarrow i(a, b) =_X i(a', b')$. It is immediate to show that $=_{A\cap B}$ satisfies the conditions of an equivalence relation and

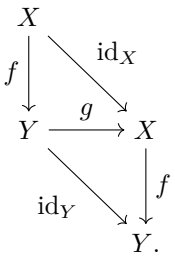
that $A \cap B$ is a set. Moreover, the assignment routine i is an embedding of $A \cap B$ into X, hence the pair $(A \cap B, i)$ is a subset of X.

The *union* $A \cup B$ of A and B is the totality defined by $z \in A \cup B :\Leftrightarrow z \in A$ or $z \in B$. If $j : A \cup B \rightsquigarrow X$ is defined by

$$j(z) := \begin{cases} i_A(z) & , z \in A \\ i_B(z) & , z \in B, \end{cases}$$

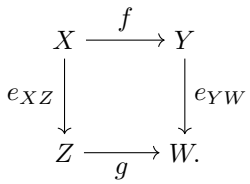
for every $z \in A \cup B$, we define $z =_{A\cup B} w :\Leftrightarrow j(z) =_X j(w)$. It is immediate to show that $=_{A\cup B}$ satisfies the conditions of an equivalence relation and that $A \cup B$ is a set. Moreover, the assignment routine j is an embedding of $A \cup B$ into X, hence the pair $(A \cup B, j)$ is a subset of X.

The *universe of sets* $\mathbb{V}_0$ is the defined totality with equality defined by $X \in \mathbb{V}_0 :\Leftrightarrow$ X is a set, and $X =_{\mathbb{V}_0} Y :\Leftrightarrow \exists_{f:X\to Y}\exists_{g:Y\to X}\big(g \circ f =_{\mathrm{id}_X} \ \& \ f \circ g = \mathrm{id}_Y\big)$



If the functions f, g realize the equality between X and Y in $\mathbb{V}_0$, we write $(f, g) : X =_{\mathbb{V}_0} Y$. It is easy to show that $X =_{\mathbb{V}_0} Y$ satisfies the conditions of an equivalence relation[5]. The defined totality with equality $\mathbb{V}_0$ is a class, since its membership condition does not reflect a construction. It is also easy to see that if $(f, g) : (A, i_A) =_{\mathcal{P}(X)} (B, i_B)$, then $(f, g) : A =_{\mathbb{V}_0} B$. Since sets and functions in BST are objects that are not reduced to one another, the next defined totality complements naturally the universe of sets $\mathbb{V}_0$ and it is proven instrumental to the formulation of dependency within CSFT. The *universe of functions* $\mathbb{V}_1$ is the defined totality with equality defined by $z \in \mathbb{V}_1 :\Leftrightarrow \exists_{X,Y\in\mathbb{V}_0}\exists_{f\in\mathbb{F}(X,Y)}\big(z := (X, Y, f)\big)$, and

$$(X, Y, f) =_{\mathbb{V}_1} (Z, W, g) :\Leftrightarrow \exists_{e_{XZ}\in\mathbb{F}(X,Z)}\exists_{e_{ZX}\in\mathbb{F}(Z,X)}\exists_{e_{YW}\in\mathbb{F}(Y,W)}\exists_{e_{WY}\in\mathbb{F}(W,Y)}$$
$$\big((e_{XZ}, e_{ZX}) : X =_{\mathbb{V}_0} Z, \ \& \ (e_{YW}, e_{WY}) : Y =_{\mathbb{V}_0} W \ \& \ e_{YW} \circ f = g \circ e_{XZ}\big)$$



If e_{XZ}, e_{ZX}, e_{YW} and e_{WY} realize the equality $(X, Y, f) =_{\mathbb{V}_1} (Z, W, g)$ in $\mathbb{V}_1$, we write

$$(e_{XZ}, e_{ZX}, e_{YW}, e_{WY}) : (X, Y, f) =_{\mathbb{V}_1} (Z, W, g).$$

[5] The defined equality on the universe $\mathbb{V}_0$ expresses that $\mathbb{V}_0$ is *univalent*, as isomorphic sets are equal in $\mathbb{V}_0$. In univalent type theory, which is MLTT extended with Voevodsky's axiom of univalence (see [36]), the existence of a pair of quasi-inverses between types A and B implies that they are equivalent in Voevodsky's sense, and by the univalence axiom, also propositionally equal. The univalence of $\mathbb{V}_0$ in CSFT is not a surprise. Already in BST the type-theoretic axiom of function extensionality is just the defined equality on the function space.

Clearly, $\mathbb{V}_1$ is a class. It is straightforward to show that $(X, Y, f) =_{\mathbb{V}_1} (Z, W, g)$ satisfies the conditions of an equivalence relation. It is also easy to see that if $(f, g) : (A, i_A) =_{\mathcal{P}(X)} (B, i_B)$, then $(f, g, \mathrm{id}_X, \mathrm{id}_X) : (A, X, i_A) =_{\mathbb{V}_1} (B, X, i_B)$.

3 Exterior union and dependent products in CSFT

The concept of a family of sets indexed by a (discrete) set was asked to be defined in [5] (Exercise 2, p. 72), and the required definition, attributed to Richman, is included in [9], (Exercise 2, p. 78), where the discreteness-hypothesis is omitted. The definition has a strong type-theoretic flavor, although Richman's motivation was categorical [6]. The concept of a (discrete) set-indexed family of sets is tacitly used in [5] in the definition of a countable product of metric spaces (see also the related comment in [9], p. 125.). We reformulate Richman's definition using the universes $\mathbb{V}_0$, $\mathbb{V}_1$ and the notion of assignment routine.

▶ **Definition 1.** *Let I be a set and $D(I)$ its diagonal. A family of sets indexed by I, or an I-family of sets, is a pair $\Lambda := (\lambda_0, \lambda_1)$, where $\lambda_0 : I \rightsquigarrow \mathbb{V}_0$ and $\lambda_1 : D(I) \rightsquigarrow \mathbb{V}_1$ are assignment routines such that for every $(i, j) \in D(I)$ we have that $\lambda_1(i, j) := (\lambda_0(i), \lambda_0(j), \lambda_{ij})$ such that for every $i \in I$ we have that $\lambda_{ii} := \mathrm{id}_{\lambda_0(i)}$, and for every $i, j, k \in I$, satisfying $i =_I j$ and $j =_I k$, the following diagram commutes*

$$\begin{array}{ccc} \lambda_0(i) & & \\ {\scriptstyle \lambda_{ij}}\downarrow & \searrow^{\lambda_{ik}} & \\ \lambda_0(j) & \xrightarrow[\lambda_{jk}]{} & \lambda_0(k). \end{array}$$

We call I the index set of the family Λ, the function λ_{ij} the transport function[7] from $\lambda_0(i)$ to $\lambda_0(j)$, and the assignment routine λ_1 the modulus of function-likeness of λ_0. If Y is a set and $\lambda_0(i) := Y$, for every $i \in I$, and $\lambda_1(i, j) := (Y, Y, \mathrm{id}_Y)$, for every $(i, j) \in D(I)$, we call Λ the constant I-family Y.

Next we see why we used the term modulus of function-likeness for the routine λ_1.

▶ Remark 2. If $\Lambda = (\lambda_0, \lambda_1)$ is an I-family of sets and $i =_I j$, then $(\lambda_{ij}, \lambda_{ji}) : \lambda_0(i) =_{\mathbb{V}_0} \lambda_0(j)$.

Proof. By Definition 1 we have that $\lambda_{ii} = \lambda_{ji} \circ \lambda_{ij}$ and $\lambda_{jj} = \lambda_{ij} \circ \lambda_{ji}$. ◀

Next we give some useful examples of set-indexed families of sets (see Proposition 9).

▶ **Definition 3.** *Let $\Lambda^{\mathbf{2}} := (\lambda_0^{\mathbf{2}}, \lambda_1^{\mathbf{2}})$, where $\lambda_0^{\mathbf{2}} : \mathbf{2} \rightsquigarrow \mathbb{V}_0$ with $\lambda_0^{\mathbf{2}}(0) := X$ and $\lambda_0^{\mathbf{2}}(1) := Y$, and $\lambda_1^{\mathbf{2}} : \{(0,0), (1,1)\} \rightsquigarrow \mathbb{V}_1$ is defined by $\lambda_1^{\mathbf{2}}(0,0) := (X, X, \mathrm{id}_X)$ and $\lambda_1^{\mathbf{2}}(1,1) := (Y, Y, \mathrm{id}_Y)$. We call $\Lambda^{\mathbf{2}}$ the $\mathbf{2}$-family of X and Y. The $\ltimes$-family of the sets $X_1, \ldots, X_n$, for every $n \geq 1$, is defined similarly. Let $\Lambda^{\mathbb{N}} := (\lambda_0^{\mathbb{N}}, \lambda_1^{\mathbb{N}})$, where $\lambda_0^{\mathbb{N}} : \mathbb{N} \rightsquigarrow \mathbb{V}_0$ with $\lambda_0^{\mathbb{N}}(n) := X_n$, and $\lambda_1^{\mathbb{N}} : \{(n, n) \mid n \in \mathbb{N}\} \rightsquigarrow \mathbb{V}_0$ is defined by $\lambda_1^{\mathbb{N}}(n, n) := (X_n, X_n, \mathrm{id}_{X_n})$, for every $n \in \mathbb{N}$. We call $\Lambda^{\mathbb{N}}$ the $\mathbb{N}$-family of $(X_n)_n$.*

Following Beeson's notation in [4], p. 44, we use the type-theoretic notation of $\sum$-types for the exterior union of a set-indexed family of sets.

---

[6] In a personal communication, Richman referred to the definition of a set-indexed family of objects of a category, given in [25], p. 18, as the source of the definition attributed to him in [9], p. 78.

[7] We draw this term from MLTT.

▶ **Definition 4.** *Let $\Lambda := (\lambda_0, \lambda_1)$ be an I-family of sets. The exterior union, or disjoint union, $\sum_{i\in I}\lambda_0(i)$ of Λ is defined by*

$$w \in \sum_{i\in I}\lambda_0(i) :\Leftrightarrow \exists_{i\in I}\exists_{x\in\lambda_0(i)}\big(w := (i,x)\big).$$

$$(i,x) =_{\sum_{i\in I}\lambda_0(i)} (j,y) :\Leftrightarrow i =_I j \ \&\ \lambda_{ij}(x) =_{\lambda_0(j)} y.$$

▶ Remark 5. The equality on $\sum_{i\in I}\lambda_0(i)$ satisfies the conditions of an equivalence relation, and $\sum_{i\in I}\lambda_0(i)$ is a set.

Proof. Let $(i,x),(j,y)$ and $(k,z) \in \sum_{i\in I}\lambda_0(i)$. Since $i =_I i$ and $\lambda_{ii} := \mathrm{id}_{\lambda_0(i)}$, we get $(i,x) =_{\sum_{i\in I}\lambda_0(i)} (i,x)$. If $(i,x) =_{\sum_{i\in I}\lambda_0(i)} (j,y)$, then $j =_I i$ and $\lambda_{ji}(y) = \lambda_{ji}(\lambda_{ij}(x)) = \lambda_{ii}(x) := \mathrm{id}_{\lambda_0(i)}(x) := x$, hence $(j,y) =_{\sum_{i\in I}\lambda_0(i)} (i,x)$. If $(i,x) =_{\sum_{i\in I}\lambda_0(i)} (j,y)$ and $(j,y) =_{\sum_{i\in I}\lambda_0(i)} (k,z)$, then from the hypotheses $i =_I j$ and $j =_I k$, we get $i =_I k$, and $\lambda_{ik}(x) = (\lambda_{jk}\circ\lambda_{ij})(x) := \lambda_{jk}(\lambda_{ij}(x)) = \lambda_{jk}(y) = z$. Clearly, the membership condition of $\sum_{i\in I}\lambda_0(i)$ reflects a construction. ◀

▶ **Definition 6.** *Let $\Lambda := (\lambda_0, \lambda_1)$ be an I-family of sets. The first projection on $\sum_{i\in I}\lambda_0(i)$ is the assignment routine $\mathtt{pr}_1(\Lambda) : \sum_{i\in I}\lambda_0(i) \rightsquigarrow I$, defined by, for every $(i,x) \in \sum_{i\in I}\lambda_0(i)$,*

$$\mathtt{pr}_1(\Lambda)(i,x) := \mathbf{pr}_1(i,x) := i.$$

We may only write $\mathtt{pr}_1$, when the family of sets Λ is clearly understood from the context.

By the definition of equality on $\sum_{i\in I}\lambda_0(i)$ we get immediately that $\mathtt{pr}_1 : \sum_{i\in I}\lambda_0(i) \to I$. At the moment, for the second projection rule $\mathtt{pr}_2(i,x) := x$, for every $(i,x) \in \sum_{i\in I}\lambda_0(i)$, we do not have a way to describe its codomain. If $\Lambda^{\mathbb{N}}$ is the $\mathbb{N}$-family of $(X_n)_n$ (Definition 3), its exterior union is by definition

$$\sum_{n\in\mathbb{N}} X_n =: \{(n,x) \mid n \in \mathbb{N} \ \&\ x \in X_n\},$$

$$(n,x) =_{\sum_{n\in\mathbb{N}} X_n} (m,y) :\Leftrightarrow n =_{\mathbb{N}} m \ \&\ x =_{X_n} y.$$

Traditionally, the countable product of this sequence of sets is defined by

$$\prod_{n\in\mathbb{N}} X_n := \left\{\phi : \mathbb{N} \to \sum_{n\in\mathbb{N}} X_n \mid \forall_{n\in\mathbb{N}}\big(\phi(n) \in X_n\big)\right\},$$

which is a rough writing of the following

$$\prod_{n\in\mathbb{N}} X_n := \left\{\phi : \mathbb{N} \to \sum_{n\in\mathbb{N}} X_n \mid \forall_{n\in\mathbb{N}}\big(\mathtt{pr}_1(\phi(n)) =_{\mathbb{N}} n\big)\right\}.$$

In the second writing the condition $\mathtt{pr}_1(\phi(n)) =_{\mathbb{N}} n$ implies that $\mathtt{pr}_1(\phi(n)) := n$, hence, if $\phi(n) := (m,y)$, then $m := n$ and $y \in X_n$. When the equality of I though, is not like that of $\mathbb{N}$, we cannot solve this problem in a satisfying way. One could define

$$\phi \in \prod_{i\in I}\lambda_0(i) :\Leftrightarrow \phi \in \mathbb{F}\left(I, \sum_{i\in I}\lambda_0(i)\right) \ \&\ \forall_{i\in I}\big(\mathtt{pr}_1(\phi(i)) := i\big).$$

This approach has the problem that the property

$$Q(\phi) :\Leftrightarrow \forall_{i\in I}\big(\mathtt{pr}_1(\phi(i)) := i\big)$$

is not necessarily extensional; let $\phi =_{\mathbb{F}\left(I,\sum_{i\in I}\lambda_0(i)\right)} \theta$ i.e., $\forall_{i\in I}\big(\phi(i) =_{\sum_{i\in I}\lambda_0(i)} \theta(i)\big)$, and suppose that $Q(\phi)$. If we fix some $i \in I$, and $\phi(i) := (i, x)$ and $\theta(i) := (j, y)$, we only get that $j =_I i$. The universe of functions $\mathbb{V}_1$ allows us to take a different approach to the definition of an arbitrary product, which, in our view, reflects accurately Bishop's formulation of dependent functions in [5], p. 65.

▶ **Definition 7.** *Let $\Lambda := (\lambda_0, \lambda_1)$ be an I-family of sets, and let $\mathbf{1} := \{x \in \mathbb{N} \mid x =_{\mathbb{N}} 0\} =: \{0\}$. A dependent function over Λ is an assignment routine $\Phi : I \rightsquigarrow \mathbb{V}_1$, where, for every $i \in I$,*

$$\Phi(i) := (\mathbf{1}, \lambda_0(i), \phi_i)$$

such that, for every $(i, j) \in D(I)$, the following diagram commutes

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_i} & \lambda_0(i) \\ {\scriptstyle \mathrm{id}_{\mathbf{1}}}\downarrow & & \downarrow{\scriptstyle \lambda_{ij}} \\ \mathbf{1} & \xrightarrow[\phi_j]{} & \lambda_0(j) \end{array}$$

Since $\phi_i : \mathbf{1} \to \lambda_0(i)$, the triple $\Phi(i)$ determines the element $\phi_i(0) \in \lambda_0(i)$. If $i =_I j$, the commutativity of the above diagram gives that $\phi_j(0) =_{\lambda_0(j)} \lambda_{ij}(\phi_i(0))$. A dependent function Φ is a function-like object i.e., $i =_I j \Rightarrow \Phi(i) =_{\mathbb{V}_1} \Phi(j)$, since $(\mathrm{id}_{\mathbf{1}}, \mathrm{id}_{\mathbf{1}}, \lambda_{ij}, \lambda_{ji}) : (\mathbf{1}, \lambda_0(i), \phi_i) =_{\mathbb{V}_1} (\mathbf{1}, \lambda_0(j), \phi_j)$. Since $\mathrm{id}_{\mathbf{1}}$ is the only function from $\mathbf{1}$ to $\mathbf{1}$, from now on we avoid mentioning it in commutative diagrams.

▶ **Definition 8.** *Let $\Lambda := (\lambda_0, \lambda_1)$ be an I-family of sets. The I-product of the family Λ is the totality $\prod_{i\in I}\lambda_0(i)$ of dependent functions over Λ equipped with the equality*

$$\Phi =_{\prod_{i\in I}\lambda_0(i)} \Theta :\Leftrightarrow \forall_{i\in I}\big(\phi_i(0) =_{\lambda_0(i)} \theta_i(0)\big)$$

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_i} & \lambda_0(i) \\ \downarrow & & \downarrow{\scriptstyle \lambda_{ii}} \\ \mathbf{1} & \xrightarrow[\theta_i]{} & \lambda_0(i) \end{array}$$

If Y is a set and Λ is the constant I-family Y, we use the notation $Y^I := \prod_{i\in I} Y$.

Clearly, the equality on $\prod_{i\in I}\lambda_0(i)$ satisfies the conditions of an equivalence relation, and $\prod_{i\in I}\lambda_0(i)$ is a set. As expected, the dependent product generalises the cartesian product.

▶ **Proposition 9.** *If $\Lambda^{\mathbf{2}}$ is the $\mathbf{2}$-family of the sets X and Y, then*

$$\prod_{i\in\mathbf{2}} \lambda_0^{\mathbf{2}}(i) =_{\mathbb{V}_0} X \times Y.$$

Proof. If $\Phi \in \prod_{i\in I}\lambda_0^{\mathbf{2}}(i)$, then $\Phi : \mathbf{2} \rightsquigarrow \mathbb{V}_1$, where $\Phi(0) := (\mathbf{1}, X, \phi_0)$ with $\phi_0 : \mathbf{1} \to X$, and $\Phi(1) := (\mathbf{1}, X, \phi_1)$ with $\phi_1 : \mathbf{1} \to Y$, such that the following diagrams commute

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_0} & X \\ \downarrow & & \downarrow{\scriptstyle \mathrm{id}_X} \\ \mathbf{1} & \xrightarrow[\phi_0]{} & X \end{array} \qquad \begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_1} & Y \\ \downarrow & & \downarrow{\scriptstyle \mathrm{id}_Y} \\ \mathbf{1} & \xrightarrow[\phi_1]{} & Y. \end{array}$$

Since this is always the case, ϕ_0, ϕ_1 are arbitrary. If $\Phi, \Theta \in \prod_{i\in I} \lambda_0^{\mathbf{2}}(i)$, then $\Phi =_{\prod_{i\in I} \lambda_0^{\mathbf{2}}(i)} \Theta$, if the following diagrams commute

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_0} & X \\ \downarrow & & \downarrow{\scriptstyle \mathrm{id}_X} \\ \mathbf{1} & \xrightarrow[\theta_0]{} & X \end{array} \qquad \begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_1} & Y \\ \downarrow & & \downarrow{\scriptstyle \mathrm{id}_Y} \\ \mathbf{1} & \xrightarrow[\theta_1]{} & Y \end{array}$$

i.e., if $\theta_0(0) =_X \phi_0(0)$ and $\theta_1(0) =_Y \phi_1(0)$. If we define $f : \prod_{i\in I} \lambda_0^{\mathbf{2}}(i) \to X \times Y$ by $f(\Phi) := (\phi_0(0), \phi_1(0))$, and $g : X \times Y \to \prod_{i\in I} \lambda_0^{\mathbf{2}}$ by $g(x, y) := \Phi_{x,y}$ with $\phi_0(0) := x$ and $\phi_1(0) := y$, it is immediate to show that $(f, g) : \prod_{i\in I} \lambda_0^{\mathbf{2}}(i) =_{\mathbb{V}_0} X \times Y$. ◀

If $\Lambda^{\mathbb{N}} := (\lambda_0^{\mathbb{N}}, \lambda_1^{\mathbb{N}})$ is the $\mathbb{N}$-family of $(X_n)_n$, and if $\Phi \in \prod_{n\in\mathbb{N}} X_n$, then, for every $n \in \mathbb{N}$, we have that $\Phi(n) := (\mathbf{1}, X_n, \phi_n)$ and the required diagram is commutative. If (X_n, ρ_n) is a metric space, for every $n \in \mathbb{N}$, Bishop's definition in [5], p. 79, of the *countable product metric* on $\prod_{n\in\mathbb{N}} X_n$ takes the form

$$\rho(\Phi, \Theta) := \sum_{n=1}^{\infty} \frac{\rho_n\big(\phi_n(0), \theta_n(0)\big)}{2^n}.$$

▶ **Proposition 10.** *If $\Lambda := (\lambda_0, \lambda_1)$ is the constant I-family Y, then $Y^I =_{\mathbb{V}_0} \mathbb{F}(I, Y)$.*

Proof. Let the assignment routine $e : Y^I \rightsquigarrow \mathbb{F}(I, Y)$ be defined by $\Phi \mapsto e(\Phi)$, and $e(\Phi)(i) := \phi_i(0)$, where $\Phi(i) := (\mathbf{1}, \lambda_0(i), \phi_i)$, for every $i \in I$. This routine is well-defined, since, if $i =_I j$, and using the equality $\lambda_{ij}(\phi_i(0)) =_{\lambda_0(j)} \phi_j(0)$, we get $e(\Phi)(i) := \phi_i(0) =_{\lambda_0(j)} \phi_j(0) := e(\Phi)(j)$, hence $e(\Phi)$ is in $\mathbb{F}(I, Y)$. The assignment routine e is also a function i..e, $\Phi =_{Y^I} \Theta \Rightarrow e(\Phi) =_{\mathbb{F}(I,Y)} e(\Theta)$, since for every $i \in I$, we have that $e(\Phi)(i) := \phi_i(0) =_{\lambda_0(i)} \theta_i(0) := e(\Theta)(i)$. Let the assignment routine $e' : \mathbb{F}(I, Y) \rightsquigarrow Y^I$ be defined by $f \mapsto e'(f)$, and $e'(f)(i) := (\mathbf{1}, Y, f_i)$, where $f_i : \mathbf{1} \to Y$ is defined by $f_i(0) := f(i)$. The assignment routine e' is a function i.e., $f =_{\mathbb{F}(I,Y)} g \Rightarrow e'(f) =_{Y^I} e'(g)$, by the equalities $f_i(0) := f(i) =_Y g(i) := g_i(0)$ and the resulting commutativity of the following diagram

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{g_i} & Y \\ \downarrow & & \downarrow{\scriptstyle \mathrm{id}_Y} \\ \mathbf{1} & \xrightarrow[f_i]{} & Y \end{array}$$

for every $i \in I$. Since $e'(f)(i) := (\mathbf{1}, Y, f_i)$, we get $e(e'(f))(i) := f_i(0) := f(i)$, hence $e \circ e' := f$. Since $e'(e(\Phi))(i) := (\mathbf{1}, Y, e(\Phi)_i)$, where $e(\Phi)_i : \mathbf{1} \to Y$ is defined by $e(\Phi)_i(0) := e(\Phi)(i) := \phi_i(0)$, we get $e(\Phi)_i := \phi_i$, and since $\Phi(i) := (\mathbf{1}, Y, \phi_i)$, for every $i \in I$, we conclude that $e'(e(\Phi)) := \Phi$. Consequently, $(e, e') : Y^I =_{\mathbb{V}_0} \mathbb{F}(I, Y)$. ◀

▶ **Definition 11.** *Let $\Lambda := (\lambda_0, \lambda_1)$ be an I-family of sets. The $\sum_{i\in I}\lambda_0(i)$-family $M_\Lambda := (\mu_0, \mu_1)$ of sets is defined by*

$$\mu_0(i,x) := \lambda_0(i),$$

$$\mu_1\big((i,x),(j,y)\big) := \big(\mu_0(i,x), \mu_0(j,y), \mu_{(i,x)(j,y)}\big) := (\lambda_0(i), \lambda_0(j), \lambda_{ij}),$$

for every $(i,x) \in \sum_{i\in I}\lambda_0(i)$ and $\big((i,x),(j,y)\big)$ in the diagonal of $\sum_{i\in I}\lambda_0(i)$. The second projection on $\sum_{i\in I}\lambda_0(i)$ is the assignment routine $\mathtt{pr}_2(\Lambda) : \sum_{i\in I}\lambda_0(i) \rightsquigarrow \mathbb{V}_1$, defined, for every $(i,x) \in \sum_{i\in I}\lambda_0(i)$, by

$$\mathtt{pr}_2(\Lambda)(i,x) := (\mathbf{1}, \lambda_0(i), \phi_{(i,x)}),$$

where $\phi_{(i,x)} : \mathbf{1} \to \lambda_0(i)$ is defined by $\phi_{(i,x)}(0) := x$. We may only write $\mathtt{pr}_2$, when the family of sets Λ is clearly understood from the context.

▶ **Proposition 12.** *If Λ and M_Λ are as in Definition 11, then*

$$\mathtt{pr}_2(\Lambda) \in \prod_{w\in\sum_{i\in I}\lambda_0(i)} \mu_0(w) := \prod_{w\in\sum_{i\in I}\lambda_0(i)} \lambda_0(\mathtt{pr}_1(w)).$$

Proof. It suffices to show that if $(i,x) =_{\sum_{i\in I}\lambda_0(i)} (j,y)$, the following diagram commutes

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_{(i,x)}} & \lambda_0(i) \\ \big\downarrow & & \big\downarrow{\scriptstyle \lambda_{ij}} \\ \mathbf{1} & \xrightarrow[\phi_{(j,y)}]{} & \lambda_0(j) \end{array}$$

By the related definitions we get $\lambda_{ij}(\phi_{(i,x)}(0)) := \lambda_{ij}(x) =_{\lambda_0(j)} y := \phi_{(j,y)}(0)$. ◀

3.1 The distributivity of $\prod$ over $\sum$

Next we prove the translation of the type-theoretic axiom of choice within CSFT (Theorem 18), or, as it was suggested to us by M. Maietti, the distributivity of $\prod$ over $\sum$[8]. For the proof of Theorem 18 we need some preparation.

▶ **Definition 13.** *Let X, Y be sets, and $R := (\rho_0, \rho_1)$ a family of sets indexed by $X \times Y$. If $x \in X$ let $\Lambda^x := (\lambda_0^x, \lambda_1^x)$, where $\lambda_0^x : Y \rightsquigarrow \mathbb{V}_0$ with $\lambda_0^x(y) := \rho_0(x,y)$, and $\lambda_1^x : D(Y) \rightsquigarrow \mathbb{V}_1$ with*

$$\lambda_1^x(y,y') := \big(\lambda_0^x(y), \lambda_0^x(y'), \lambda_{yy'}^x\big) := \big(\rho_0(x,y), \rho_0(x,y'), \rho_{(x,y)(x,y')}\big),$$

for every $y \in Y$ and every $(y,y') \in D(Y)$, respectively. Let also $M := (\mu_0, \mu_1)$, where $\mu_0 : X \rightsquigarrow \mathbb{V}_0$ with $\mu_0(x) := \sum_{y\in Y}\rho_0(x,y)$, and $\mu_1 : D(X) \rightsquigarrow \mathbb{V}_1$ with

$$\mu_1(x,x') := \big(\mu_0(x), \mu_0(x'), \mu_{xx'}\big) := \left(\sum_{y\in Y}\rho_0(x,y), \sum_{y\in Y}\rho_0(x',y), \mu_{xx'}\right),$$

[8] We would like to E. Palmgren for pointing to us that such a distributivity holds in every locally cartesian closed category. In [38] it is mentioned that this fact is generally attributed to Martin-Löf and his work [24]. For a proof see [2].

for every $x \in X$ and every $(x, x') \in D(X)$, respectively. For every $(y, u) \in \sum_{y\in Y} \rho_0(x, y)$, let

$$\mu_{xx'} : \sum_{y\in Y} \rho_0(x, y) \to \sum_{y\in Y} \rho_0(x', y) \quad \& \quad \mu_{xx'}(y, u) := \big(y, \rho_{(x,y)(x',y)}(u)\big).$$

▶ **Lemma 14.** *The pairs $\Lambda^x := (\lambda^x_0, \lambda^x_1)$ and $M := (\mu_0, \mu_1)$ in Definition 13 are families of sets indexed by Y and X, respectively.*

Proof. Since by hypothesis R is an $X \times Y$-family of sets, we get

$$\lambda^x_1(y, y) := \big(\rho_0(x, y), \rho_0(x, y), \rho_{(x,y)(x,y)}\big) := \big(\rho_0(x, y), \rho_0(x, y), \mathrm{id}_{\rho_0(x,y)}\big),$$

and the commutativity of the left diagram

$$\begin{array}{ccc} \lambda^x_0(y) & & \\ {\scriptstyle \lambda^x_{yy'}}\downarrow & \searrow^{\lambda^x_{yy''}} & \\ \lambda^x_0(y') & \xrightarrow[\lambda^x_{y'y''}]{} & \lambda^x_0(y'') \end{array} \qquad \begin{array}{ccc} \rho_0(x, y) & & \\ {\scriptstyle \rho_{(x,y)(x,y')}}\downarrow & \searrow^{\rho_{(x,y)(x,y'')}} & \\ \rho_0(x, y') & \xrightarrow[\rho_{(x,y')(x,y'')}]{} & \rho_0(x, y'') \end{array}$$

is by definition the known commutativity of the right diagram. Similarly,

$$\mu_1(x, x) := \big(\mu_0(x), \mu_0(x), \mu_{xx}\big) := \left(\sum_{y\in Y} \rho_0(x, y), \sum_{y\in Y} \rho_0(x, y), \mu_{xx}\right),$$

where $\mu_{xx} : \sum_{y\in Y} \rho_0(x, y) \to \sum_{y\in Y} \rho_0(x, y)$ is defined by

$$\mu_{xx}(y, u) := \big(y, \rho_{(x,y)(x,y)}(u)\big) := \big(y, \mathrm{id}_{\rho_0(x,y)}(u)\big) := (y, u),$$

for every $(y, u) \in \sum_{y\in Y} \rho_0(x, y)$. For the commutativity of the left diagram

$$\begin{array}{ccc} \mu_0(x) & & \\ {\scriptstyle \mu_{xx'}}\downarrow & \searrow^{\mu_{xx''}} & \\ \mu_0(x') & \xrightarrow[\mu_{x'x''}]{} & \mu_0(x'') \end{array} \qquad \begin{array}{ccc} \rho_0(x, y) & & \\ {\scriptstyle \rho_{(x,y)(x',y)}}\downarrow & \searrow^{\rho_{(x,y)(x'',y)}} & \\ \rho_0(x', y) & \xrightarrow[\rho_{(x',y)(x'',y)}]{} & \rho_0(x'', y) \end{array}$$

we use the known commutativity of the right diagram, since

$$\begin{aligned} \mu_{x'x''}\big(\mu_{xx'}(y, u)\big) &:= \mu_{x'x''}\big(y, \rho_{(x,y)(x',y)}(u)\big)\big) \\ &:= \big(y, \rho_{(x',y)(x'',y)}(\rho_{(x,y)(x',y)}(u))\big) \\ &:= \big(y, \rho_{(x,y)(x'',y)}(u)\big) \\ &:= \mu_{xx''}(y, u), \end{aligned}$$

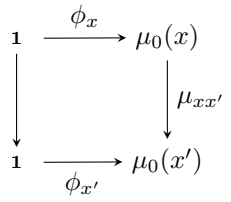
for every $(y, u) \in \sum_{y\in Y} \rho(x, y)$. ◀

▶ **Lemma 15.** *Let $R := (\rho_0, \rho_1)$, $\Lambda^x := (\lambda^x_0, \lambda^x_1)$ and $M := (\mu_0, \mu_1)$ be the families of sets of Definition 13. If $\Phi \in \prod_{x\in X} \mu_0(x)$, then Φ generates a function $f_\Phi : X \to Y$.*

Proof. By definition, $\Phi : X \rightsquigarrow \mathbb{V}_1$, where, for every $x \in X$,

$$\Phi(x) := (\mathbf{1}, \mu_0(x), \phi_x) := \big(\mathbf{1}, \sum_{y\in Y} \rho_0(x, y), \phi_x\big),$$

where $\phi_x : \mathbf{1} \to \sum_{y\in Y} \rho_0(x, y)$. We define the assignment routine $f_\Phi : X \rightsquigarrow Y$ by the rule $f_\Phi(x) := \mathtt{pr}_1(\phi_x(0))$, for every $x \in X$. Next we show that the routine f_Φ is a function. Let $x =_X x'$. By the commutativity of the following diagram



we have that, if $\phi_x(0) := (y, u)$, for some $y \in Y$ and $u \in \rho_0(x, y)$, then

$$\mu_{xx'}(\phi_x(0)) := \mu_{xx'}(y, u) := \big(y, \rho_{(x,y)(x',y)}(u)\big) =_{\sum_{y \in Y} \rho_0(x',y)} \phi_{x'}(0),$$

hence, since $\mathtt{pr}_1$ is a function, we get

$$f(x') := \mathtt{pr}_1(\phi_{x'}(0)) =_Y \mathtt{pr}_1\big(y, \rho_{(x,y)(x',y)}(u)\big) := y := \mathtt{pr}_1(\phi_x(0)) := f(x). \qquad \blacktriangleleft$$

▶ **Lemma 16.** *Let $R := (\rho_0, \rho_1)$, $\Lambda^x := (\lambda_0^x, \lambda_1^x)$ and $M := (\mu_0, \mu_1)$ be the families of sets of Definition 13. If $f : X \to Y$, let $N^f := (\nu_0^f, \nu_1^f)$, where $\nu_0^f : X \rightsquigarrow \mathbb{V}_0$ and $\nu_1^f : D(X) \rightsquigarrow \mathbb{V}_1$ are defined by*

$$\nu_0^f(x) := \rho_0(x, f(x)),$$

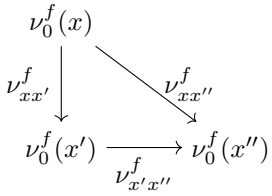
$$\nu_1^f(x, x') := \big(\nu_0^f(x), \nu_0^f(x'), \nu_{xx'}^f\big) := \big(\rho_0(x, f(x)), \rho_0(x', f(x')), \rho_{(x,f(x))(x',f(x'))}\big),$$

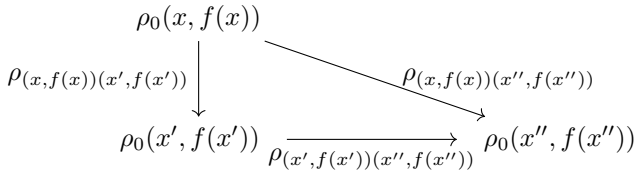
for every $x \in X$ and every $(x, x') \in D(X)$, respectively, then N^f is an X-family of sets.

Proof. Since by hypothesis R is an $X \times Y$-family of sets, we get

$$\begin{aligned}\nu_1^f(x, x) &:= \big(\rho_0(x, f(x)), \rho_0(x, f(x)), \rho_{(x,f(x))(x,f(x))}\big)\\ &:= \big(\rho_0(x, f(x)), \rho_0(x, f(x)), \mathrm{id}_{\rho_0(x,f(x))}\big).\end{aligned}$$

Since by Lemma 15 f_Φ is a function, the commutativity of the left diagram





is by definition the known commutativity of the right diagram. $\blacktriangleleft$

▶ **Lemma 17.** *Let $R := (\rho_0, \rho_1)$ be the family of sets in Definition 13, and $N^f := (\nu_0^f, \nu_1^f)$ the family of sets defined in Lemma 16. If $\Xi := (\xi_0, \xi_1)$, where $\xi_0 : \mathbb{F}(X, Y) \rightsquigarrow \mathbb{V}_0$ and $\xi_1 : D(\mathbb{F}(X, Y)) \rightsquigarrow \mathbb{V}_1$ are defined by*

$$\xi_0(f) := \prod_{x \in X} \nu_0^f(x) := \prod_{x \in X} \rho_0(x, f(x))$$

$$\xi_1(f, f') := \big(\xi_0(f), \xi_0(f'), \xi_{ff'}\big),$$

where

$$\xi_{ff'} : \prod_{x \in X} \rho_0(x, f(x)) \to \prod_{x \in X} \rho_0(x, f'(x))$$

is defined by

$$\xi_{ff'}(H)(x) := \big(\nVdash, \rho_0(x, f'(x)), h'_x\big),$$

$$h'_x(0) := \rho_{(x,f(x))(x,f'(x))}\big(h_x(0)\big),$$

and

$$H(x) := \big(\mathbf{1}, \nu_0^f(x), h_x\big) := \big(\mathbf{1}, \rho_0(x, f(x)), h_x\big)$$

for every $H \in \prod_{x\in X} \rho_0(x, f(x))$ *and* $x \in X$*, then* Ξ *is a family of sets indexed by* $\mathbb{F}(X, Y)$*.*

Proof. First we show that if $f =_{\mathbb{F}(X,Y)} f'$, then

$$\xi_{ff'}(H) \in \prod_{x\in X} \rho_0(x, f'(x)) := \prod_{x\in X} \nu_0^{f'}(x),$$

by showing that if $x =_X x'$, then the commutativity of the left diagram

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{h_x} & \rho_0(x, f(x)) \\ \downarrow & & \downarrow{\scriptstyle \nu^f_{xx'}} \\ \mathbf{1} & \xrightarrow[h_{x'}]{} & \rho_0(x', f(x')) \end{array} \qquad \begin{array}{ccc} \mathbf{1} & \xrightarrow{h'_x} & \rho_0(x, f'(x)) \\ \downarrow & & \downarrow{\scriptstyle \nu^{f'}_{xx'}} \\ \mathbf{1} & \xrightarrow[h'_{x'}]{} & \rho_0(x', f'(x')) \end{array}$$

implies the commutativity of the right one. By definition we have that

$$\begin{aligned} \nu^{f'}_{xx'}\big(h'_x(0)\big) &:= \nu^{f'}_{xx'}\Big(\rho_{(x,f(x))(x,f'(x))}\big(h_x(0)\big)\Big) \\ &:= \rho_{(x,f'(x))(x',f'(x'))}\Big(\rho_{(x,f(x))(x,f'(x))}\big(h_x(0)\big)\Big) \\ &=_{\rho_0(x',f'(x'))} \quad \rho_{(x,f(x)(x',f'(x'))}\big(h_x(0)\big). \end{aligned}$$

since the pairs $(x, f(x)), (x, f'(x))$ and $(x', f'(x'))$ are equal in $X \times Y$, by the hypotheses $x =_X x'$ and $f =_{\mathbb{F}(X,Y)} f'$. Moreover, by the commutativity of the left diagram above we get

$$h_{x'}(0) =_{\rho_0(x',f(x'))} \nu^f_{xx'}\big(h_x(0)\big) =_{\rho_0(x',f(x'))} \rho_{(x,f(x))(x',f(x'))}\big(h_x(0)\big),$$

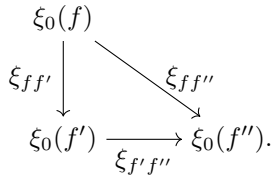
hence,

$$\begin{aligned} h'_{x'}(0) &:= \rho_{(x',f(x')(x',f'(x')}\big(h_{x'}(0)\big) \\ &=_{\rho_0(x',f(x'))} \quad \rho_{(x',f(x')(x',f'(x')}\Big(\rho_{(x,f(x))(x',f(x'))}\big(h_x(0)\big)\Big) \\ &=_{\rho_0(x',f(x'))} \quad \rho_{(x,f(x)(x',f'(x'))}\big(h_x(0)\big), \end{aligned}$$

and consequently, $\nu^{f'}_{xx'}\big(h'_x(0)\big) =_{\rho_0(x',f(x'))} h'_{x'}(0)$. Next we show that ξ_1 satisfies the properties of Definition 1. By definition $\xi_{ff}(H)(x) := \big(\mathbf{1}, \rho_0(x, f(x)), h'_x\big)$, where

$$h'_x(0) := \rho_{(x,f(x))(x,f(x))}\big(h_x(0)\big) := \mathrm{id}_{\rho_0(x,f(x))}\big(h_x(0)\big) := h_x(0),$$

hence $\xi_{ff}(H) := H$, and since H is arbitrary, we get $\xi_{ff} := \mathrm{id}_{\xi_0(f)}$. Finally, if $f =_{\mathbb{F}(X,Y)} f' =_{\mathbb{F}(X,Y)} f''$, we show the commutativity of the following diagram



If $H \in \xi_0(f)$, we show $\xi_{ff''}(H) =_{\xi_0(f'')} \xi_{f'f''}(\xi_{ff'}(H))$ i.e.,

$$\xi_{ff''}(H) \;\; =_{\prod_{x\in X}\rho_0(x,f''(x))} \;\; \xi_{f'f''}(\xi_{ff'}(H)).$$

By definition we have that $[\xi_{f'f''}(\xi_{ff'}(H))](x) := (\mathbf{1}, \rho_0(x, f''(x)), {h''}_x)$, where

$${h''}_x(0) := \rho_{(x,f'(x))(x,f''(x))}\big({h'}_x(0)\big).$$

Since $\xi_{ff'}(H)(x) := (\mathbf{1}, \rho_0(x, f'(x)), {h'}_x)$, where ${h'}_x(0) := \rho_{(x,f(x))(x,f'(x))}\big(h_x(0)\big)$, we get

$${h''}_x(0) := \rho_{(x,f'(x))(x,f''(x))}\Big(\rho_{(x,f(x))(x,f'(x))}\big(h_x(0)\big)\Big) = \rho_{(x,f(x))(x,f''(x))}\big(h_x(0)\big) := \tau_x(0),$$

where $\xi_{ff''}(H)(x) := (\mathbf{1}, \rho_0(x, f''(x)), \tau_x)$, with $\tau_x(0) := \rho_{(x,f(x))(x,f''(x))}\big(h_x(0)\big)$, and since $x \in X$ is arbitrary, the required commutativity is shown. ◀

▶ **Theorem 18** (Distributivity of $\prod$ over $\sum$)**.** *Let X, Y be sets, and $R := (\rho_0, \rho_1)$, $\Lambda^x := (\lambda_0^x, \lambda_1^x)$, $M := (\mu_0, \mu_1)$ the families of sets of Definition 13. If*

$$\Phi \in \prod_{x\in X} \mu_0(x) := \prod_{x\in X}\sum_{y\in Y} \rho_0(x, y),$$

there is $\Theta_\Phi \in \prod_{x\in X} \nu_0^{f_\Phi}(x)$, where $f_\Phi : X \to Y$ is defined in Lemma 15, and

$$\big(f_\Phi, \Theta_\Phi\big) \in \sum_{f\in\mathbb{F}(X,Y)}\prod_{x\in X} \nu_0^f(x) := \sum_{f\in\mathbb{F}(X,Y)}\prod_{x\in X} \rho_0(x, f(x)).$$

Moreover, the assignment routine

$$\mathtt{ac} : \prod_{x\in X}\sum_{y\in Y} \rho_0(x, y) \;\rightsquigarrow\; \sum_{f\in\mathbb{F}(X,Y)}\prod_{x\in X} \rho_0(x, f(x))$$

$$\mathtt{ac}(\Phi) := \big(f_\Phi, \Theta_\Phi\big)$$

is a function.

Proof. By Proposition 12 we have that

$$\mathtt{pr}_2(\Lambda^x) \in \prod_{w\in\sum_{y\in Y}\lambda_0^x(y)} \lambda_0^x(\mathtt{pr}_1(w)) := \prod_{w\in\sum_{y\in Y}\rho_0(x,y)} \rho_0(x, \mathtt{pr}_1(w)),$$

where, if $(y, u) \in \sum_{y\in Y} \rho_0(x, y)$, then $\mathtt{pr}_2(\Lambda^x)(y, u) := \big(\mathbf{1}, \rho_0(x, y), \sigma_{(y,u)}\big)$, and $\sigma_{(y,u)} : \mathbf{1} \to \rho_0(x, y)$ is defined by $\sigma_{(y,u)}(0) := u$. We define the assignment routine $\Theta_\Phi : X \rightsquigarrow \mathbb{V}_1$ by

$$\Theta_\Phi(x) := \big(\mathbf{1}, \nu_0^{f_\Phi}(x), \theta_x\big) := \big(\mathbf{1}, \rho_0(x, f_\Phi(x)), \theta_x\big),$$

where $\theta_x : \mathbf{1} \to \rho_0(x, f_\Phi(x))$ is defined by $\theta_x(0) := \sigma_{(y,u)}(0) := u$, and $\phi_x(0) := (y, u) := (f_\Phi(x), u)$. Since $(y, u) := \phi_x(0) \in \sum_{y\in Y} \rho_0(x, y)$, we have that $u \in \rho_0(x, y) := \rho_0(x, f_\Phi(x))$. In order to show that $\Theta_\Phi \in \prod_{x\in A} \nu_0^{f_\Phi}(x) := \prod_{x\in A} \rho_0(x, f_\Phi(x))$, we need to show, for $x =_X x'$, the commutativity of the following diagram

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\theta_x} & \rho_0(x, f_\Phi(x)) \\ \downarrow & & \downarrow \nu^{f_\Phi}_{xx'} \\ \mathbf{1} & \xrightarrow[\theta_{x'}]{} & \rho_0(x', f_\Phi(x')). \end{array}$$

Since $\Phi \in \prod_{x\in X} \mu_0(x) := \prod_{x\in X} \sum_{y\in Y} \rho_0(x, y)$, we have the commutativity of the diagram

$$\begin{array}{ccc} \mathbf{1} & \xrightarrow{\phi_x} & \sum_{y\in Y} \rho_0(x, y) \\ \downarrow & & \downarrow \mu_{xx'} \\ \mathbf{1} & \xrightarrow[\phi_{x'}]{} & \sum_{y\in Y} \rho_0(x', y), \end{array}$$

where by Definition 13 this commutativity becomes

$$\mu_{xx'}(\phi_x(0)) := \mu_{xx'}(y, u) := \big(y, \rho_{(x,y)(x',y)}(u)\big) := \big(y, \rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big)$$

$$=_{\sum_{y\in Y} \rho_0(x',y)} \quad \phi_{x'}(0) := (y', u') := (f_\Phi(x'), u').$$

Since the equality

$$\big(y, \rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big) \quad =_{\sum_{y\in Y} \rho_0(x',y)} \quad (y', u')$$

is by definition the equality

$$\big(y, \rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big) \quad =_{\sum_{y\in Y} \lambda^{x'}_0(y)} \quad (y', u'),$$

we have that $y =_Y y'$ and

$$\lambda^{x'}_{yy'}\big(\rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big) =_{\lambda^{x'}_0(y')} \ u',$$

while by the definition of $\lambda^{x'}_{yy'}$ and since $\lambda^{x'}_0(y') := \rho_0(x', y')$ we get

$$\rho_{(x',y)(x',y')}\big(\rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big) =_{\rho_0(x',y')} \ u'$$

i.e.,

$$\rho_{(x',f_\Phi(x))(x',f_\Phi(x'))}\big(\rho_{(x,f_\Phi(x))(x',f_\Phi(x))}(u)\big) =_{\rho_0(x',y')} \ u'.$$

By the commutativity of the following diagram

$$\begin{array}{ccc} \rho_0(x, f_\Phi(x)) & & \\ \downarrow \rho_{(x,f_\Phi(x))(x',f_\Phi(x))} & \searrow^{\rho_{(x,f_\Phi(x))(x',f_\Phi(x'))}} & \\ \rho_0(x', f_\Phi(x)) & \xrightarrow[\rho_{(x',f_\Phi(x))(x',f_\Phi(x'))}]{} & \rho_0(x', f_\Phi(x')) \end{array}$$

we get

$$\rho_{(x,f_\Phi(x))(x',f_\Phi(x'))}(u) =_{\rho_0(x',y')} \; u',$$

and the required commutativity of the diagram for Θ_Φ is shown as follows:

$$\nu^{f_\Phi}_{xx'}(\theta_x(0)) := \nu^{f_\Phi}_{xx'}(\sigma_{(y,u)}(0)) := \nu^{f_\Phi}_{xx'}(u_0) := \rho_{(x,f_\Phi(x))(x',f_\Phi(x'))}(u) =_{\rho_0(x',y')} \; u' := \theta_{x'}(0).$$

Next we show that $\mathtt{ac}$ is a function i.e., $\Phi =_{\prod_{x\in X}\mu_0(x)} \Phi' \Rightarrow \mathtt{ac}(\Phi) =_{\sum_{f\in\mathbb{F}(X,Y)}\xi_0(f)} \mathtt{ac}(\Phi')$. If

$$\Phi(x) := (\mathbf{1}, \mu_0(x), \phi_x) := (\mathbf{1}, \sum_{y\in Y}\rho_0(x,y), \phi_x),$$

$$\Phi'(x) := (\mathbf{1}, \mu_0(x), {\phi'}_x) := (\mathbf{1}, \sum_{y\in Y}\rho_0(x,y), {\phi'}_x),$$

the hypothesis $\Phi =_{\prod_{x\in X}\mu_0(x)} \Phi'$ is reduced to $\phi_x(0) =_{\mu_0(x)} {\phi'}_x(0)$, for every $x \in X$. By definition the equality

$$\big(f_\Phi, \Theta_\Phi\big) \;\; =_{\sum_{f\in\mathbb{F}(X,Y)}\xi_0(f)} \;\; \big(f_{\Phi'}, \Theta_{\Phi'}\big)$$

is reduced to $f_\Phi =_{\mathbb{F}(X,Y)} f_{\Phi'}$ and

$$\xi_{f_\Phi f_{\Phi'}}(\Theta_\Phi) \;\; =_{\xi_0(f_{\Phi'})} \;\; \Theta_{\Phi'} \;\; :\Leftrightarrow \xi_{f_\Phi f_{\Phi'}}(\Theta_\Phi) \;\; =_{\prod_{x\in X}\rho_0(x,f_{\Phi'}(x))} \;\; \Theta_{\Phi'}.$$

If $x \in X$, then

$$f_\Phi(x) := \mathtt{pr}_1(\phi_x(0)) =_Y \mathtt{pr}_1({\phi'}_x(0)) := f_{\Phi'}(x),$$

hence, since $x \in X$ is arbitrary, $f_\Phi =_{\mathbb{F}(X,Y)} f_{\Phi'}$. By definition $\Phi(x) := (\mathbf{1}, \sum_{y\in Y}\rho_0(x,y), \phi_x)$ and $\Theta_\Phi(x) := (\mathbf{1}, \rho_0(x, f_\Phi(x)), \theta_x)$, where $\theta_x(0) := \sigma_{(y,u)}(0) := u$, and $\phi_x(0) := (y,u) := (f_\Phi(x), u)$. Similarly, $\Phi'(x) := (\mathbf{1}, \sum_{y\in Y}\rho_0(x,y), {\phi'}_x)$ and $\Theta_{\Phi'}(x) := (\mathbf{1}, \rho_0(x, f_{\Phi'}(x)), {\theta'}_x)$, where ${\theta'}_x(0) := \sigma_{(y',u')}(0) := u'$, and ${\phi'}_x(0) := (y',u') := (f_{\Phi'}(x), u')$. Moreover,

$$\xi_{f_\Phi f_{\Phi'}}(\Theta_\Phi)(x) := (\mathbf{1}, \rho_0(x, f_{\Phi'}(x)), {h'}_x),$$

$${h'}_x(0) := \rho_{(x,f_\Phi(x))(x,f_{\Phi'}(x))}\big(\theta_x(0)\big) := \rho_{(x,f_\Phi(x))(x,f_{\Phi'}(x))}(u).$$

By definition, we need to show that, for every $x \in X$,

$${\theta'}_x(0) \;\; =_{\rho_0(x,f_{\Phi'}(x))} \;\; {h'}_x(0) :\Leftrightarrow u' \;\; =_{\rho_0(x,f_{\Phi'}(x))} \;\; \rho_{(x,f_\Phi(x))(x,f_{\Phi'}(x))}(u).$$

Since

$$\phi_x(0) =_{\mu_0(x)} {\phi'}_x(0) :\Leftrightarrow \phi_x(0) =_{\sum_{y\in Y}\rho_0(x,y)} {\phi'}_x(0) :\Leftrightarrow (f_\Phi(x), u) =_{\sum_{y\in Y}\lambda^x_0(y)} (f_{\Phi'}(x), u'),$$

we get

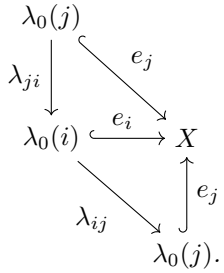
$$\lambda^x_{yy'}(u) =_{\rho_0(x,y')} u' :\Leftrightarrow \rho_{(x,f_\Phi(x))(x,f_{\Phi'}(x))}(u) =_{\rho_0(x,y')} u',$$

which is exactly what we need to show. ◀

4 Interior union and dependent products in CSFT

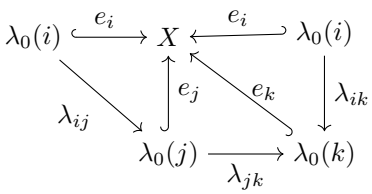
Next we formulate Bishop's definition of a set-indexed family of subsets, given in [5], p. 65, in analogy to our definition of a set-indexed family of sets.

▶ **Definition 19.** *Let X and I be sets. A family of subsets of X indexed by I is a triple $\lambda := (\lambda_0, \sigma_1, \lambda_1)$, where $\lambda_0 : I \rightsquigarrow \mathbb{V}_0$, and $\sigma_1 : I \rightsquigarrow \mathbb{V}_1$, such that, for every $i \in I$, we have that $\sigma_1(i) := (\lambda_0(i), X, e_i)$ and e_i is an embedding of $\lambda_0(i)$ into X. Moreover, $\lambda_1 : D(I) \rightsquigarrow \mathbb{V}_1$ is called the modulus of function-likeness of λ_0, and for every $i \in I$ it satisfies $\lambda_{ii} := \mathrm{id}_{\lambda_0(i)}$, while for every $(i, j) \in D(I)$ it satisfies $(\lambda_{ij}, \lambda_{ji}) : \lambda_0(i) =_{\mathcal{P}(X)} \lambda_0(j)$ i.e., the following inner diagrams commute*



▶ Remark 20. If $\lambda := (\lambda_0, \sigma_1, \lambda_1)$ is an I-family of subsets of X, then $\Lambda_\lambda := (\lambda_0, \lambda_1)$ is an I-family of sets.

Proof. Let $i =_I j =_I k$. If $a \in \lambda_0(i)$, by the commutativity of the following inner diagrams



and omitting the subscripts in the following equalities, we have that

$$e_k\big(\lambda_{jk}(\lambda_{ij}(a))\big) = e_j(\lambda_{ij}(a)) = e_i(a) = e_k\big(\lambda_{ik}(a)\big),$$

hence $\lambda_{jk}(\lambda_{ij}(a)) = \lambda_{ik}(a)$, and since $a \in \lambda_0(i)$ is arbitrary, we get $\lambda_{jk} \circ \lambda_{ij} = \lambda_{ik}$. ◀

▶ **Definition 21.** *Let $\lambda := (\lambda_0, \sigma_0, \lambda_1)$ be an I-family of subsets of X. The interior union of λ is the totality $\bigcup_{i\in I} \lambda_0(i)$ defined by*

$$z \in \bigcup_{i\in I} \lambda_0(i) :\Leftrightarrow \exists_{i\in I} \exists_{x\in\lambda_0(i)} \big(z := (i, x)\big).$$

Let the assignment routine $\varepsilon : \bigcup_{i\in I} \lambda_0(i) \rightsquigarrow X$ be defined by $\varepsilon(i, x) := e_i(x)$, for every $(i, x) \in \bigcup_{i\in I} \lambda_0(i)$, where $e_i : \lambda_0(i) \hookrightarrow X$ is the embedding of $\lambda_0(i)$ into X, for every $i \in I$. The equality on $\bigcup_{i\in I} \lambda_0(i)$ is defined by

$$(i, x) =_{\bigcup_{i\in I} \lambda_0(i)} (j, y) :\Leftrightarrow \varepsilon(i, x) =_X \varepsilon(j, y).$$

It is immediate to show that $(i,x) =_{\bigcup_{i\in I}\lambda_0(i)} (j,y)$ satisfies the conditions of an equivalence relation, and $\bigcup_{i\in I}\lambda_0(i)$ is a set. Moreover, the assignment routine ε is an embedding of $\bigcup_{i\in I}\lambda_0(i)$ into X, hence the pair $\big(\bigcup_{i\in I}\lambda_0(i),\varepsilon\big)$ is a subset of X. Note that $\sum_{i\in I}\lambda_0(i)$ and $\bigcup_{i\in I}\lambda_0(i)$ have the same membership formula, but different equalities. The equality of $\sum_{i\in I}\lambda_0(i)$ is determined *externally* by the transport function λ_{ij}, while the equality of $\bigcup_{i\in I}\lambda_0(i)$ is determined *internally* by the embeddings e_i, e_j.

▶ **Definition 22.** *Let $\lambda := (\lambda_0,\sigma_0,\lambda_1)$ be an I-family of subsets of X. A dependent function over λ is*[9] *a dependent function over Λ_λ. Based on Definition 7, and using a superscript to emphasize that we deal with a family of subsets, we denote their set by $\prod^{X}_{i\in I}\lambda_0(i)$.*

Next we define the intersection of a set-indexed family of subsets (see also [5], p. 65).

▶ **Definition 23.** *Let $\lambda := (\lambda_0,\sigma_1,\lambda_1)$ be an I-family of subsets of X, where I is inhabited by some element i_0. The intersection $\bigcap_{i\in I}\lambda_0(i)$ of λ is the totality defined by*

$$\Phi \in \bigcap_{i\in I}\lambda_0(i) :\Leftrightarrow \Phi \in \prod^{X}_{i\in I}\lambda_0(i) \;\&\; \forall_{i,i'\in I}\big(e_i(\phi_i(0)) =_X e_{i'}(\phi_{i'}(0))\big),$$

where, for every $i\in I$, $\Phi(i) := (\mathbf{1},\lambda_0(i),\phi_i)$ and $\sigma_1(i) := (\lambda_0(i),X,e_i)$. Let the assignment routine $e : \bigcap_{i\in I}\lambda_0(i) \rightsquigarrow X$ be defined by $e(\Phi) := e_{i_0}\big(\phi_{i_0}(0)\big)$. If $\Phi,\Theta\in\bigcap_{i\in I}\lambda_0(i)$, we define

$$\Phi =_{\bigcap_{i\in I}\lambda_0(i)} \Theta :\Leftrightarrow e(\Phi) =_X e(\Theta).$$

It is immediate to show that $\Phi =_{\bigcap_{i\in I}\lambda_0(i)} \Theta$ satisfies the conditions of an equivalence relation, and $\bigcap_{i\in I}\lambda_0(i)$ is a set. Moreover, the assignment routine e is an embedding of $\bigcap_{i\in I}\lambda_0(i)$ into X, hence the pair $\big(\bigcap_{i\in I}\lambda_0(i),e\big)$ is a subset of X. As expected, Definition 21 is the family-version of the definition of $A\cup B$, and Definition 23 is the family-version of the definition of $A\cap B$. Working as in Proposition 9, we get the following.

▶ Remark 24. Let $A,B\in\mathcal{P}(X)$, and let $\lambda^{\mathbf{2}} := (\lambda^{\mathbf{2}}_0,\sigma^{\mathbf{2}}_1,\lambda^{\mathbf{2}}_1)$ be a $\mathbf{2}$-family of subsets of X, where $\lambda^{\mathbf{2}}_0$, $\lambda^{\mathbf{2}}_1$ are defined as in Definition 3, $\sigma^{\mathbf{2}}_1(0) := (A,X,i_A)$, and $\sigma^{\mathbf{2}}_1(1) := (B,X,i_B)$. Then

$$\bigcup_{i\in\mathbf{2}}\lambda_0(i) =_{\mathcal{P}(X)} A\cup B \quad\&\quad \bigcap_{i\in\mathbf{2}}\lambda_0(i) =_{\mathcal{P}(X)} A\cap B.$$

5 Concluding remarks

There are many issues regarding the relation between BST and CSFT that, due to lack of space, cannot be elaborated here. E.g., in the literature of constructive mathematics (see e.g., [10]) the powerset $\mathcal{P}(X)$ of a set X is treated as a set. Bishop's comment in [5], p. 68, on the existence of a map (i.e., a function) from the complemented subsets of X to $\mathcal{P}(X)$ seems to support such a view. In his definition though, of a set-indexed family of subsets in [5], p. 65, Bishop is careful to use the notion of a rule (an assignment routine) which only behaves like a function. As Bishop himself explains in [8], p. 67, on the occasion of the

[9] The definition of $\prod_{i\in I}\lambda_0(i)$, given in [9], p. 70, as the set $\big\{f : I\to\bigcup_{i\in I}\lambda_0(i) \mid \forall_{i\in I}\big(f(i)\in\lambda_0(i)\big)\big\}$ is not compatible with the precise definition of $\bigcup_{i\in I}\lambda_0(i)$, given previously in the same page, and it is not included in [5].

precise definition of a measure space, one must rewrite appropriately the material in [5], in order to be "comfortably" formalised. Such an appropriate rewriting explains our use of the universes $\mathbb{V}_0$ and $\mathbb{V}_1$. In our view, the totality $\mathbb{V}_0$ is implicit in Bishop's formulation in [5], p. 72, regarding the definition of a set-indexed family of sets. There he writes about "... a rule which assigns to each t in a discrete set T a set $\lambda(t)$". Similarly, the universe $\mathbb{V}_1$ is just a way to rewrite appropriately notions of rules that assign elements of an index set to sets and functions between them with certain properties.

A variation of Definition 1 is the constructive version of the direct spectrum over a directed set (see [14], p. 420). If I is a set and $i \preceq j$ an extensional and transitive relation on $I \times I$, let $\preceq (I) := \{(i,j) \in I \times I \mid i \preceq j\}$. An *$I$-transitive family of sets with respect to* $\preceq$ is a pair $\Lambda^{\preceq} := (\lambda_0, \lambda_1^{\preceq})$, where $\lambda_0 : I \rightsquigarrow \mathbb{V}_0$, $\lambda_1^{\preceq} :\preceq (I) \rightsquigarrow \mathbb{V}_1$ where $\lambda_1^{\preceq}(i,j) := (\lambda_0(i), \lambda_0(j), \lambda_{ij}^{\preceq})$, such that for every $i,j,k \in I$ with $i \preceq j$ and $j \preceq k$ the following diagram commutes

$$\begin{array}{ccc} \lambda_0(i) & & \\ {\scriptstyle \lambda_{ij}^{\preceq}}\downarrow & \searrow^{\lambda_{ik}^{\preceq}} & \\ \lambda_0(j) & \xrightarrow[\lambda_{jk}^{\preceq}]{} & \lambda_0(k). \end{array}$$

If $(I, \preceq)$ is a directed preorder i.e., $i \preceq j$ is irreflexive, transitive, and directed i.e., $\forall_{i,j\in I}\exists_{k\in I}\big(i \preceq k \;\&\; j \preceq k\big)$, we call $\Lambda^{\preceq}$ a *direct family of sets* over I. One can define on $\sum_{i\in I} \lambda_0(i)$ the following equality

$$(i,x) =_{\sum_{i\in I}\lambda_0(i)} (j,y) :\Leftrightarrow \exists_{k\in I}\big(i \preceq k \;\&\; j \preceq k \;\&\; \lambda_{ik}^{\preceq}(x) =_{\lambda_0(k)} \lambda_{jk}^{\preceq}(y)\big).$$

We study the direct families of sets and their corresponding dependent sums and dependent products in [34].

In [5], p. 65, Bishop defined an *I-set* of subsets of a set X as an I-family $\lambda := (\lambda_0, \sigma_0, \lambda_1)$ of subsets of X such that $\forall_{i,j\in I}\big(\lambda_0(i) =_{\mathcal{P}(X)} \lambda_0(j) \Rightarrow i =_I j\big)$ i.e., the converse to $i =_I j \Rightarrow \lambda_0(i) =_{\mathcal{P}(X)} \lambda_0(j)$ also holds. A basic property of such a family is that functions on the index set I generate functions on the set $\lambda_0 I$ defined by $z \in \lambda_0 I :\Leftrightarrow \exists_{i\in I}\big(z := \lambda_0(i)\big)$, equipped with the equality $\lambda_0(i) =_{\lambda_0 I} \lambda_0(j) :\Leftrightarrow \lambda_0(i) =_{\mathcal{P}(X)} \lambda_0(j)$. This property is crucial to the definition of measure space, given in [9], p. 282 (see Bishop's comment in [8], p. 67).

A general feature of BST is its harmonious relationship with the topology of Bishop spaces (see [30]). If F_i is a Bishop topology on $\lambda_0(i)$, for every $i \in I$, in [34] we define, with the use of the notion of a least Bishop topology, a canonical Bishop topology on the exterior union $\sum_{i\in I} \lambda_0(i)$ and the dependent product $\prod_{i\in I} \lambda_0(i)$. A precise formulation of the least Bishop topology relies on the study of inductively defined sets within Bishop's system BISH* and its expected reconstruction within an appropriate extension CSFT* of CSFT. The development of CSFT*, the extension of CSFT with inductive definitions of sets using rules with countably many premisses, is, hopefully, future work.

References

1 Peter Aczel and Michael Rathjen. Notes on Constructive Set Theory. Technical report, Institut Mittag–Leffler Preprint, 2000.

2 Steven Awodey. Axiom of Choice and Excluded Middle in Categorical Logic. *Bulletin of Symbolic Logic*, 1:344, 1995.

3 Michael J. Beeson. Formalizing constructive mathematics: Why and how? In *Constructive Mathematics*, pages 146–190. Springer Berlin Heidelberg, 1981. doi:10.1007/bfb0090733.

4 Michael J. Beeson. *Foundations of Constructive Mathematics*. Springer Berlin Heidelberg, 1985. doi:10.1007/978-3-642-68952-9.

5 Errett Bishop. *Foundations of Constructive Analysis*. Mcgraw-Hill, 1967.

6 Errett Bishop. A General Lamguage. *unpublished manuscript*, 1968(9)?

7 Errett Bishop. How to Compile Mathematics into Algol. *unpublished manuscript*, 1968(9)?

8 Errett Bishop. Mathematics as a Numerical Language. In *Intuitionism and Proof Theory: Proceedings of the Summer Conference at Buffalo N.Y. 1968*, pages 53–71. Elsevier, 1970. doi:10.1016/s0049-237x(08)70740-7.

9 Errett Bishop and Douglas S. Bridges. *Constructive Analysis*. Grundlehren der mathematischen Wissenschaften. Springer Berlin Heidelberg, 1985.

10 Douglas S. Bridges and Fred Richman. *Varieties of constructive mathematics*. Cambridge University Press, 1987.

11 Thierry Coquand, Peter Dybjer, Erik Palmgren, and Anton Setzer. *Type-theoretic Foundations of Constructive Mathematics*. book draft, 2005.

12 Thierry Coquand and Henrik Persson. Integrated Development of Algebra in Type Theory. *manuscript*, 1998.

13 Thierry Coquand and Arnaud Spiwack. Towards Constructive Homological Algebra in Type Theory. In *Towards Mechanized Mathematical Assistants*, pages 40–54. Springer Berlin Heidelberg, 2007. doi:10.1007/978-3-540-73086-6_4.

14 James Dugundji. *Topology*. Allyn and Bacon series in advanced mathematics. Allyn and Bacon, 1966.

15 Robert Lee Constable et. al. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

16 Solomon Feferman. A language and axioms for explicit mathematics. In *Algebra and Logic*, pages 87–139. Springer Berlin Heidelberg, 1975. doi:10.1007/bfb0062852.

17 Solomon Feferman. Constructive Theories of Functions and Classes. In *Studies in Logic and the Foundations of Mathematics*, pages 159–224. Elsevier, 1979. doi:10.1016/s0049-237x(08)71625-2.

18 Harvey Friedman. Set Theoretic Foundations for Constructive Analysis. *The Annals of Mathematics*, 105(1):1, January 1977. doi:10.2307/1971023.

19 Newcomb Greenleaf. Liberal constructive set theory. In *Constructive Mathematics*, pages 213–240. Springer Berlin Heidelberg, 1981.

20 Hajime Ishihara and Erik Palmgren. Quotient topologies in constructive set theory and type theory. *Annals of Pure and Applied Logic*, 141(1-2):257–265, August 2006. doi:10.1016/j.apal.2005.11.005.

21 Maria Emilia Maietti. A minimalist two-level foundation for constructive mathematics. *Annals of Pure and Applied Logic*, 160(3):319–354, September 2009. doi:10.1016/j.apal.2009.01.006.

22 Per Martin-Löf. *Notes on Constructive Mathematics*. Almqvist and Wiksell, 1968.

23 Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. In *Logic Colloquium '73, Proceedings of the Logic Colloquium*, pages 73–118. Elsevier, 1975. doi:10.1016/s0049-237x(08)71945-1.

24 Per Martin-Löf. *Intuitionistic type theory*, volume 1 of *Studies in Proof Theory*. Bibliopolis, 1984. Notes by Giovanni Sambin of a series of lectures given in Padua, June 1980.

25 Ray Mines, Fred Richman, and Wim Ruitenburg. *A course in constructive algebra*. Springer-Verlag, 1988.

26 John Myhill. Constructive set theory. *Journal of Symbolic Logic*, 40(3):347–382, September 1975. doi:10.2307/2272159.

27 Erik Palmgren. Bishop's set theory. *Slides from TYPES Summer School 2005, Gothenburg*, 2005.

28 Erik Palmgren. *Lecture Notes on Type Theory*. manuscript, 2014.

29 Erik Palmgren and Olov Wilander. Constructing categories and setoids of setoids in type theory. *Logical Methods in Computer Science*, Volume 10, Issue 3, 2014. doi:10.2168/LMCS-10(3:25)2014.

30 Iosif Petrakis. *Constructive topology of Bishop spaces*. PhD thesis, Ludwig-Maximilians-University, Munich, 2015.

31 Iosif Petrakis. A constructive function-theoretic approach to topological compactness. In *Proceedings of the 31st Annual ACM/IEEE Symposium on Logic in Computer Science - LICS '16*. ACM Press, 2016. doi:10.1145/2933575.2933582.

32 Iosif Petrakis. The Urysohn Extension Theorem for Bishop Spaces. In *Logical Foundations of Computer Science*, pages 299–316. Springer International Publishing, 2016. doi:10.1007/978-3-319-27683-0_21.

33 Iosif Petrakis. Constructive uniformities of pseudometrics and Bishop topologies. *Journal of Logic and Analysis*, 11:FT2:1–44, 2019.

34 Iosif Petrakis. Direct spectra of Bishop spaces and their limits. *arXiv:1907.03273*, 2019.

35 Iosif Petrakis. *Constructive Set and Function Theory*. habilitation, Ludwig-Maximilians-University, Munich, in preparation, 2020.

36 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. Institute for Advanced Study, Princeton, 2013.

37 Giovanni Sambin. *The Basic Picture: Structures for Constructive Topology*. Oxford University Press, in press, 2019.

38 Tilo Wiklund. Locally cartesian closed categories, coalgebras, and containers. Technical report, Uppsala Universitet, U.U.D.M Project Report, 2013.

Semantic Subtyping for Non-Strict Languages

Tommaso Petrucciani
DIBRIS, Università di Genova, Italy
IRIF, Université Paris Diderot, France

Giuseppe Castagna
CNRS, IRIF, Université Paris Diderot, France

Davide Ancona
DIBRIS, Università di Genova, Italy

Elena Zucca
DIBRIS, Università di Genova, Italy

Abstract

Semantic subtyping is an approach to define subtyping relations for type systems featuring union and intersection type connectives. It has been studied only for strict languages, and it is unsound for non-strict semantics. In this work, we study how to adapt this approach to non-strict languages: in particular, we define a type system using semantic subtyping for a functional language with a call-by-need semantics. We do so by introducing an explicit representation for divergence in the types, so that the type system distinguishes expressions that are results from those which are computations that might diverge.

2012 ACM Subject Classification Software and its engineering → Functional languages

Keywords and phrases Semantic subtyping, non-strict semantics, call-by-need, union types, intersection types

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.4

Related Version `https://arxiv.org/abs/1810.05555`

1 Introduction

Semantic subtyping is a powerful framework which allows language designers to define subtyping relations for rich languages of types – including union and intersection types – that can express precise properties of programs. However, it has been developed for languages with call-by-value semantics and, in its current form, it is unsound for non-strict languages. We show how to design a type system which keeps the advantages of semantic subtyping while being sound for non-strict languages (more specifically, for call-by-need semantics).

1.1 Semantic subtyping

Union and intersection types can be used to type several language constructs – from branching and pattern matching to function overloading – very precisely. However, they make it challenging to define a subtyping relation that behaves precisely and intuitively.

Semantic subtyping is a technique to do so, studied by Frisch, Castagna, and Benzaken [20] for types given by:

$$t ::= b \mid t \to t \mid t \times t \mid t \vee t \mid t \wedge t \mid \neg t \mid \mathbb{0} \mid \mathbb{1} \qquad \text{where } b ::= \mathsf{Int} \mid \mathsf{Bool} \mid \cdots$$

Types include constructors – basic types b, arrows, and products – plus union $t \vee t$, intersection $t \wedge t$, negation (or complementation) $\neg t$, and the bottom and top types $\mathbb{0}$ and $\mathbb{1}$ (actually, $t_1 \wedge t_2$ and $\mathbb{1}$ can be defined respectively as $\neg(\neg t_1 \vee \neg t_2)$ and $\neg\mathbb{0}$). The grammar above is interpreted *coinductively* rather than inductively, thus allowing infinite type expressions

that correspond to recursive types. Subtyping is defined by giving an interpretation $[\![\cdot]\!]$ of types as sets and defining $t_1 \leq t_2$ as the inclusion of the interpretations, that is, $t_1 \leq t_2$ is defined as $[\![t_1]\!] \subseteq [\![t_2]\!]$. Intuitively, we can see $[\![t]\!]$ as the set of values that inhabit t in the language. By interpreting union, intersection, and negation as the corresponding operations on sets and by giving appropriate interpretations to the other constructors, we ensure that subtyping will satisfy all commutative and distributive laws we expect: for example, $(t_1 \times t_2) \vee (t'_1 \times t'_2) \leq (t_1 \vee t'_1) \times (t_2 \vee t'_2)$ or $(t \to t_1) \wedge (t \to t_2) \leq t \to (t_1 \wedge t_2)$.

This relation is used in [20] to type a call-by-value language featuring higher-order functions, data constructors and destructors (pairs), and a typecase construct which models runtime type dispatch and acts as a form of pattern matching. Functions can be recursive and are explicitly typed; their type can be an intersection of arrow types, describing overloaded behaviour. A simple example of an overloaded function is

```
let f x = if (x is Int) then (x + 1) else not(x)
```

which tests whether its argument `x` is of type Int and in this case returns its successor, its negation otherwise. This function can be given the type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, which signifies that it has both type $\mathsf{Int} \to \mathsf{Int}$ and type $\mathsf{Bool} \to \mathsf{Bool}$: the two types define its two possible behaviours depending on the outcome of the test (and, thus, on the type of the input). This is done in [20] by explicitly annotating the whole function definition. Using notation for typecases from [20]: $\mathsf{let}\ f : (\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) = \lambda x.\, (x \in \mathsf{Int})\ ?\ (x + 1) : (\mathsf{not}\ x)$. The type deduced for this function is $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool})$, but it can also be given the type $(\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$: the latter type states that the function can be applied to both integers and booleans and that its result is either an integer or a boolean. This latter type is less precise than the intersection, since it loses the correlation between the types of the argument and of the result. Accordingly, the semantic definition of subtyping ensures $(\mathsf{Int} \to \mathsf{Int}) \wedge (\mathsf{Bool} \to \mathsf{Bool}) \leq (\mathsf{Int} \vee \mathsf{Bool}) \to (\mathsf{Int} \vee \mathsf{Bool})$.

The work of [20] has been extended to treat more language features, including parametric polymorphism [11, 12, 14], type inference [13], and gradual typing [10] and adapted to SMT solvers [6]. It has been used to type object-oriented languages [1, 16], XML queries [9], NoSQL languages [5], and scientific languages [27]. It is also at the basis of the definition of $\mathbb{C}$Duce, an XML-processing functional programming language with union and intersection types [4]. However, only strict evaluation had been considered, until now.

1.2 Semantic subtyping in lazy languages

Our work started as an attempt to design a type system for the Nix Expression Language [17], an untyped, purely functional, and lazily evaluated language for Unix/Linux package management. Since Nix is untyped, some programming idioms it encourages require advanced type system features to be analyzed properly. Notably, the possibility of writing functions that use type tests to have an overloaded-like behaviour made intersection types and semantic subtyping a good fit for the language. However, existing semantic subtyping relations are unsound for non-strict semantics; this was already observed in [20] and no adaptation has been proposed later. Here we describe our solution to define a type system based on semantic subtyping which is sound for a non-strict language. In particular, we consider a call-by-need variant of the language studied in [20].

Current semantic subtyping systems are unsound for non-strict semantics because of the way they deal with the bottom type $\mathbb{0}$, which corresponds to the empty set of values ($[\![\mathbb{0}]\!] = \varnothing$). The intuition is that a reducible expression e can be safely given a type t only if all results (i.e., values) it can return are of type t. Accordingly, $\mathbb{0}$ can only be assigned

to expressions that are statically known to diverge (i.e., that never return a result). For example, the ML expression $\mathsf{let\ rec\ f\ x} = \mathsf{f\ x\ in\ f\ ()}$ can be given type $\mathbb{0}$. Let us use $\bar{e}$ to denote any diverging expression that, like this, can be given type $\mathbb{0}$. Consider the following typing derivations, which are valid in current semantic subtyping systems (π_2 projects the second component of a pair).

$$\dfrac{\dfrac{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Int}}{\vdash (\bar{e}, 3) : \mathbb{0} \times \mathsf{Bool}}\,[\simeq]}{\vdash \pi_2\,(\bar{e}, 3) : \mathsf{Bool}} \qquad \dfrac{\dfrac{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Int}}{\vdash \lambda x.\,3 : \mathbb{0} \to \mathsf{Bool}}\,[\simeq] \qquad \vdash \bar{e} : \mathbb{0}}{\vdash (\lambda x.\,3)\,\bar{e} : \mathsf{Bool}}$$

Note that both $\pi_2\,(\bar{e}, 3)$ and $(\lambda x.\,3)\,\bar{e}$ diverge in call-by-value semantics (since $\bar{e}$ must be evaluated first), while they both reduce to 3 in call-by-name or call-by-need. The derivations are therefore sound for call-by-value, while they are clearly unsound with non-strict evaluation.

Why are these derivations valid? The crucial steps are those marked with $[\simeq]$, which convert between types that have the same interpretation; $\simeq$ denotes this equivalence relation. With semantic subtyping, $\mathbb{0} \times \mathsf{Int} \simeq \mathbb{0} \times \mathsf{Bool}$ holds because all types of the form $\mathbb{0} \times t$ are equivalent to $\mathbb{0}$ itself: none of these types contains any value (indeed, product types are interpreted as Cartesian products and therefore the product with the empty set is itself empty). It can appear more surprising that $\mathbb{0} \to \mathsf{Int} \simeq \mathbb{0} \to \mathsf{Bool}$ holds. We interpret a type $t_1 \to t_2$ as the set of functions which, on arguments of type t_1, either diverge or return results in type t_2. Since there is no argument of type $\mathbb{0}$ (because, in call-by-value, arguments are always values), all types of the form $\mathbb{0} \to t$ are equivalent (they all contain every well-typed function).

1.3 Our approach

The intuition behind our solution is that, with non-strict semantics, it is not appropriate to see a type as the set of the values that have that type. In a call-by-value language, operations like application or projection occur on values: thus, we can identify two types (and, in some sense, the expressions they type) if they contain (and their expressions may produce) the same values. In non-strict languages, though, operations also occur on partially evaluated results: these, like $(\bar{e}, 3)$ in our example, can contain diverging sub-expressions below their top-level constructor.

As a result, it is unsound, for example, to type $(\bar{e}, 3)$ as $\mathbb{0} \times \mathsf{Int}$, since we have that $\mathbb{0} \times \mathsf{Int}$ and $\mathbb{0} \times \mathsf{Bool}$ are equivalent. It is also unsound to have subtyping rules for functions which assume implicitly that every argument will eventually be a value.

One approach to solve this problem would be to change the interpretation of $\mathbb{0}$ so that it is non-empty. However, the existence of types with an empty interpretation is important for the internal machinery of semantic subtyping. Notably, the decision procedure for subtyping relies on them (checking whether $t_1 \leq t_2$ holds is reduced to checking whether the type $t_1 \wedge \neg t_2$ is empty). Therefore, we keep the interpretation $[\![\mathbb{0}]\!] = \varnothing$, but we change the type system so that this type is *never* derivable, not even for diverging expressions. We keep it as a purely "internal" type useful to describe subtyping, but never used to type expressions.

We introduce instead a separate type $\bot$ as the type of diverging expressions. This type is non-empty but disjoint from the types of constants, functions, and pairs: $[\![\bot]\!]$ is a singleton whose unique element represents divergence. Introducing the type $\bot$ means that we track termination in types. In particular, we distinguish two classes of types: those that are disjoint from $\bot$ (for example, Int, $\mathsf{Int} \to \mathsf{Bool}$, or $\mathsf{Int} \times \mathsf{Bool}$) and those that include $\bot$ (since the interpretation of $\bot$ is a singleton, no type can contain a proper subset of it). Intuitively, the

former correspond to computations that are guaranteed to terminate: for example, Int is the type of terminating expressions producing an integer result. Conversely, the types of diverging expressions must always contain $\bot$ and, as a result, they can always be written in the form $t \vee \bot$, for some type t. Subtyping verifies $t \leq t \vee \bot$ for any t: this ensures that a terminating expression can always be used when a possibly diverging one is expected. This subdivision of types suggests that $\bot$ is used to approximate the set of diverging well-typed expressions: an expression whose type contains $\bot$ is an expression that *may* diverge. Actually, the type system we propose performs a rather gross approximation. We derive "terminating types" (i.e., subtypes of $\neg\bot$) only for expressions that are already results and cannot be reduced: constants, functions, or pairs. Applications and projections, instead, are always typed by assuming that they might diverge. The typing rules are written to handle and propagate the $\bot$ type. For example, we type applications using the following rule.

$$\frac{\Gamma \vdash e_1 : (t' \to t) \vee \bot \qquad \Gamma \vdash e_2 : t'}{\Gamma \vdash e_1\, e_2 : t \vee \bot}$$

This rule allows the expression e_1 to be possibly diverging: we require it to have the type $(t' \to t) \vee \bot$ instead of the usual $t' \to t$ (but an expression with the latter type can always be subsumed to have the former type). We type the whole application as $t \vee \bot$ to signify that it can diverge even if the codomain t does not include $\bot$, since e_1 can diverge.

This system avoids the problems we have seen with semantic subtyping: no expression can be assigned the empty type, which was the type on which subtyping had incorrect behaviour. The new type $\bot$ does not cause the same problems because $[\![\bot]\!]$ is non-empty. For example, the type of expressions like $(\bar{e}, 3)$ – where $\bar{e}$ is diverging – is now $\bot \times \mathsf{Int}$. This type is not equivalent to $\bot \times \mathsf{Bool}$: indeed, the two interpretations are different because the interpretation of types includes an element ($[\![\bot]\!]$) to represent divergence.

Typing all applications as possibly diverging – even very simple ones like $(\lambda x.\, 3)\, e$ – is a very coarse approximation which can seem unsatisfactory. We could try to amend the rule to say that if e_1 has type $t' \to t$, then $e_1\, e_2$ has type t instead of $t \vee \bot$. However, we prefer to keep the simpler rules since they achieve our goal of giving a sound type system that still enjoys most benefits of semantic subtyping.

An advantage of the simpler system is that it allows us to treat $\bot$ as an internal type that does not need to be written explicitly by programmers. Since the language is explicitly typed, if $\bot$ were to be treated more precisely, programmers would presumably need to include it or exclude it explicitly from function signatures. This would make the type system significantly different from conventional ones where divergence is not explicitly expressed in the types. In the present system, instead, we can assume that programmers annotate programs using standard set-theoretic types and $\bot$ is introduced only behind the scenes and, thus, is transparent to programmers.

We define this type system for a call-by-need variant of the language studied in [20], and we prove its soundness in terms of progress and subject reduction.

The choice of call-by-need rather than call-by-name stems from the behaviour of semantic subtyping on intersections of arrow types. Our type system would actually be unsound for call-by-name if the language were extended with constructs that can reduce non-deterministically to different answers. For example, the expression $\mathsf{rnd}(t)$ of [20] that returns a random value of type t could not be added while keeping soundness. This is because in call-by-name, if such an expression is duplicated, each occurrence could reduce differently; in call-by-need, instead, its evaluation would be shared. Intersection and union types make the type system precise enough to expose this difference. In the absence of such non-deterministic

constructs, call-by-name and call-by-need can be shown to be observationally equivalent, so that soundness should hold for both; however, call-by-need also simplifies the technical work to prove soundness.

We show an example of this, though we will return on this point later. Consider the following derivation, where $\bar{e}$ is an expression of type $\mathsf{Int} \vee \mathsf{Bool}$.

$$\dfrac{\dfrac{\dfrac{x\colon \mathsf{Int} \vdash (x,x)\colon \mathsf{Int}\times\mathsf{Int} \qquad x\colon \mathsf{Bool} \vdash (x,x)\colon \mathsf{Bool}\times\mathsf{Bool}}{\vdash \lambda x.\,(x,x)\colon (\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})}}{\vdash \lambda x.\,(x,x)\colon \mathsf{Int}\vee\mathsf{Bool}\to(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})}\,[\leq] \qquad \vdash \bar{e}\colon \mathsf{Int}\vee\mathsf{Bool}}{\vdash (\lambda x.\,(x,x))\,\bar{e}\colon (\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})}$$

In a system with intersection types, the function $\lambda x.\,(x,x)$ can be given the type $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$ because it has both arrow types (in practice, the function will have to be annotated with the intersection). Then, the step marked with $[\leq]$ is allowed because, in semantic subtyping, $(\mathsf{Int}\to\mathsf{Int}\times\mathsf{Int})\wedge(\mathsf{Bool}\to\mathsf{Bool}\times\mathsf{Bool})$ is a subtype of $(\mathsf{Int}\vee\mathsf{Bool})\to((\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool}))$ (in general, $(t_1\to t_1')\wedge(t_2\to t_2')\leq t_1\vee t_2\to t_1'\vee t_2'$). Therefore, the application $(\lambda x.\,(x,x))\,\bar{e}$ is well-typed with type $(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})$. In call-by-name, it reduces to $(\bar{e},\bar{e})$: therefore, for the system to satisfy subject reduction, we must be able to type $(\bar{e},\bar{e})$ with the type $(\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool})$ too. But this type is intuitively unsound for $(\bar{e},\bar{e})$ if each occurrence of $\bar{e}$ could reduce independently and non-deterministically either to an integer or to a boolean. Using a typecase we can actually exhibit a term that breaks subject reduction.

There are several ways to approach this problem. We could change the type system or the subtyping relation so that $\lambda x.\,(x,x)$ cannot be given the type $(\mathsf{Int}\vee\mathsf{Bool})\to((\mathsf{Int}\times\mathsf{Int})\vee(\mathsf{Bool}\times\mathsf{Bool}))$. However, this would curtail the expressive power of intersection types as used in the semantic subtyping approach. We could instead assume explicitly that the semantics is deterministic. In this case, the typing would not be unsound intuitively, but a proof of subject reduction would be difficult: we should give a complex union disjunction rule to type $(\bar{e},\bar{e})$. We choose instead to consider a call-by-need semantics because it solves both problems. With call-by-need, non-determinism poses no difficulty because of sharing. We still need a union disjunction rule, but it is simpler to state since we only need it to type the let bindings which represent shared computations.

1.4 Contributions

The main contribution of this work is the development of a type system for non-strict languages based on semantic subtyping; to our knowledge, this had not been studied before.

Although the idea of our solution is simple – to track divergence – its technical development is far from trivial. Our work highlights how a type system featuring union and intersection types is sensitive to the difference between strict and non-strict semantics and also, in the presence of non-determinism, to that between call-by-name and call-by-need. This shows once more how union and intersection types can express very fine properties of programs. Our main technical contribution is the description of sound typing for let bindings – a construct peculiar to most of the formalizations of call-by-need semantics – in the presence of union types. Finally, our work shows how to integrate the $\bot$ type, which is an explicit representation for divergence, in a semantic subtyping system. It can thus also be seen as a first step towards the definition of a type system based on semantic subtyping that performs a non-trivial form of termination analysis.

1.5 Related work

Previous work on semantic subtyping does not discuss non-strict semantics. Castagna and Frisch [8] describe how to add a type constructor $\mathsf{lazy}(t)$ to semantic subtyping systems, but this is meant just to have lazily constructed expressions within a call-by-value language.

Many type systems for functional languages – like the simply-typed λ-calculus or Hindley-Milner typing – are sound for both strict and non-strict semantics. However, difficulties similar to ours are found in work on refinement types. Vazou et al. [23] study how to adapt refinement types for Haskell. Their types contain logical predicates as refinements: e.g., the type of positive integers is $\{\, v\colon \mathsf{Int} \mid v > 0 \,\}$. They observe that the standard approach to typechecking in these systems – checking implication between predicates with an SMT solver – is unsound for non-strict semantics. In their system, a type like $\{\, v\colon \mathsf{Int} \mid \mathsf{false} \,\}$ is analogous to $\mathbb{0}$ in our system insofar as it is not inhabited by any value. These types can be given to diverging expressions, and their introduction into the environment causes unsoundness. To avoid this problem, they stratify types, with types divided in diverging and non-diverging ones. This corresponds in a way to our use of a type $\bot$ in types of possibly diverging expressions. As for ours, their type system can track termination to a certain extent. Partial correctness properties can be verified even without precise termination analysis. However, with their kind of analysis (which goes beyond what is expressible with set-theoretic types) there is a significant practical benefit to tracking termination more precisely. Hence, they also study how to check termination of recursive functions.

The notion of a stratification of types to keep track of divergence can also be found in work of a more theoretical strain. For instance, in [15] it is used to model partial functions in constructive type theory. This stratification can be understood as a monad for partiality, as it is treated in [7]. Our type system can also be seen, intuitively, as following this monadic structure. Notably, the rule for applications in a sense lifts the usual rule for application in this partiality monad. Injection in this monad is performed implicitly by subtyping via the judgment $t \leq t \vee \bot$. However, we have not developed this intuition formally.

The fact that a type system with union and intersection types can require changes to account for non-strict semantics is also remarked in work on refinement types. Dunfield and Pfenning [19, p. 8, footnote 3] notice how a union elimination rule cannot be used to eliminate unions in function arguments if arguments are passed by name: this is analogous to the aforementioned difficulties which led to our choice of call-by-need (their system uses a dedicated typing rule for what our system handles by subtyping). Dunfield [18, Section 8.1.5] proposes as future work to adapt a subset of the type system he considers (of refinement types for a call-by-value effectful language) to call-by-name. He notes some of the difficulties and advocates studying call-by-need as a possible way to face them. In our work we show, indeed, that a call-by-need semantics can be used to have the type system handle union and intersection types expressively without requiring complex rules.

Finally, Vouillon [24] – drawing on earlier work with Melliès [25] on interpreting types as sets of terms – studies the subtyping relation induced by such an interpretation for systems with union types. Many concerns raised in his work parallel ours. He remarks that some subtyping rules are only sound for specific calculi (e.g., only for call-by-value or only for deterministic semantics), while others are sound for large classes of calculi. He defines subtyping avoiding the rules of the first kind to have a relation which is more robust to language extensions or modifications than semantic subtyping as we use it (though, in doing so, he does not capture fully the set-theoretic intuition for strict languages). He also remarks how union elimination is problematic for non-deterministic call-by-name semantics. His interpretation of types as sets of terms is more adapted to describing non-strict semantics than

the semantic-subtyping approach of interpreting types as sets of values. However, his system does not account for negation types, that we include and interpret as set complementation: this would probably be challenging to integrate into his theory.

1.6 Outline

Our presentation proceeds as follows. In Section 2, we define the types and the subtyping relation which we use in our type system. In Section 3, we define the language we study, its syntax, and its operational semantics. In Section 4, we present the type system; we state the result of soundness for it and outline the main lemmas required to prove it; we also complete the discussion about why we chose a call-by-need semantics. In Section 5, we study the relation between the interpretation of types used to define subtyping and the expressions that are definable in the language; we show how we can look for a more precise interpretation. In Section 6 we conclude and point out more directions for future work.

For space reasons, some auxiliary definitions and results, as well as the proofs of the results we state, are omitted and can all be found in the extended version available online [22].

2 Types and subtyping

We begin by describing in more detail the types and the subtyping relation of our system.

In order to define types, we first fix two countable sets: a set $\mathcal{C}$ of *language constants* (ranged over by c) and a set $\mathcal{B}$ of *basic types* (ranged over by b). For example, we can take constants to be booleans and integers: $\mathcal{C} = \{\mathsf{true}, \mathsf{false}, 0, 1, \text{-}1, \ldots\}$. $\mathcal{B}$ might then contain Bool and Int; however, we also assume that, for every constant c, there is a "singleton" basic type which corresponds to that constant alone (for example, a type for true, which will be a subtype of Bool). We assume that a function $\mathbb{B} : \mathcal{B} \to \mathcal{P}(\mathcal{C})$ assigns to each basic type the set of constants of that type and that a function $b_{(\cdot)} : \mathcal{C} \to \mathcal{B}$ assigns to each constant c a basic type b_c such that $\mathbb{B}(b_c) = \{c\}$.

▶ **Definition 2.1** (Types). *The set $\mathcal{T}$ of* types *is the set of terms t coinductively produced by the following grammar*

$$t ::= \bot \mid b \mid t \times t \mid t \to t \mid t \vee t \mid \neg t \mid \mathbb{0}$$

and which satisfy two additional constraints: (1) regularity*: the term must have a finite number of different sub-terms;* (2) contractivity*: every infinite branch must contain an infinite number of occurrences of the product or arrow type constructors.*

We introduce the abbreviations $t_1 \wedge t_2 \stackrel{\text{def}}{=} \neg(\neg t_1 \vee \neg t_2)$, $t_1 \setminus t_2 \stackrel{\text{def}}{=} t_1 \wedge (\neg t_2)$, and $\mathbb{1} \stackrel{\text{def}}{=} \neg\mathbb{0}$. We refer to b, $\times$, and $\to$ as *type constructors*, and to $\vee$, $\neg$, $\wedge$, and $\setminus$ as *type connectives*.

The regularity condition is necessary only to ensure the decidability of the subtyping relation. Contractivity, instead, is crucial because it excludes terms which do not have a meaningful interpretation as types or sets of values: for instance, the trees satisfying the equations $t = t \vee t$ (which gives no information on which values are in it) or $t = \neg t$ (which cannot represent any set of values). Contractivity also ensures that the binary relation $\triangleright \subseteq \mathcal{T}^2$ defined by $t_1 \vee t_2 \triangleright t_i$ and $\neg t \triangleright t$ is Noetherian (that is, strongly normalizing). This gives an induction principle on $\mathcal{T}$ that we will use without further reference to the relation (e.g., in Definition 2.3). This induction principle allows us to apply the induction hypothesis below type connectives (union and negation), but not below type constructors (product and arrow). As a consequence of contractivity, types cannot contain infinite unions or intersections.

In the semantic subtyping approach we give an interpretation of types as sets; this interpretation is used to define the subtyping relation in terms of set containment. We want to see a type as the set of the values of the language that have that type. However, this set of values cannot be used directly to define the interpretation, because of a problem of circularity. Indeed, in a higher-order language, values include well-typed λ-abstractions; hence to know which values inhabit a type we need to have already defined the type system (to type λ-abstractions), which depends on the subtyping relation, which in turn depends on the interpretation of types. To break this circularity, types are actually interpreted as subsets of a set $\mathcal{D}$, an *interpretation domain*, which is not the set of values, though it corresponds to it intuitively (in [20], a correspondence is also shown formally: we return to this in Section 5). We use the following domain which includes an explicit representation for divergence.

▶ **Definition 2.2** (Interpretation domain). *The* interpretation domain $\mathcal{D}$ *is the set of finite terms* d *produced inductively by the following grammar*

$$d ::= \bot \mid c \mid (d, d) \mid \{(d, d_\Omega), \dots, (d, d_\Omega)\} \qquad\qquad d_\Omega ::= d \mid \Omega$$

where c *ranges over the set* $\mathcal{C}$ *of constants and where* Ω *is such that* $\Omega \notin \mathcal{D}$.

The elements of $\mathcal{D}$ correspond, intuitively, to the results of the evaluation of expressions. The element $\bot$ stands for divergence. Expressions can produce as results constants or pairs of results, so we include both in $\mathcal{D}$. For example, a result can be a pair of a terminating computation returning true and a diverging computation: we represent this by $(\mathsf{true}, \bot)$. Finally, in a higher-order language, the result of a computation can be a function. Functions are represented in this model by finite relations of the form $\{(d^1, d^1_\Omega), \dots, (d^n, d^n_\Omega)\}$, where Ω (which is not in $\mathcal{D}$) can appear in second components to signify that the function fails (i.e., evaluation is stuck) on the corresponding input. This constant Ω is used to ensure that $\mathbb{1} \to \mathbb{1}$ is not a supertype of all function types: if we used d instead of d_Ω, then every well-typed function could be subsumed to $\mathbb{1} \to \mathbb{1}$ and, therefore, every application could be given the type $\mathbb{1}$, indenpendently from the type of its argument (see Section 4.2 of [20] for details). The restriction to *finite* relations is standard in semantic subtyping [20]; we say more about it in Section 5.

We define the interpretation $[\![t]\!]$ of a type t so that it satisfies the following equalities, where $\mathcal{D}_\Omega = \mathcal{D} \cup \{\Omega\}$ and where $\mathcal{P}_{\text{fin}}$ denotes the restriction of the powerset to finite subsets:

$$[\![\bot]\!] = \{\bot\} \qquad [\![b]\!] = \mathbb{B}(b) \qquad [\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$$
$$[\![t_1 \to t_2]\!] = \left\{ R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \;\middle|\; \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \right\}$$
$$[\![t_1 \vee t_2]\!] = [\![t_1]\!] \cup [\![t_2]\!] \qquad [\![\neg t]\!] = \mathcal{D} \setminus [\![t]\!] \qquad [\![\mathbb{0}]\!] = \varnothing$$

We cannot take the equations above directly as an inductive definition of $[\![\cdot]\!]$ because types are not defined inductively but coinductively. Therefore we give the following definition, which validates these equalities and which uses the aforementioned induction principle on types and structural induction on $\mathcal{D}$.

▶ **Definition 2.3** (Set-theoretic interpretation of types). *We define a binary predicate* $(d_\Omega : t)$ *("the element* d_Ω *belongs to the type* t*"), where* $d_\Omega \in \mathcal{D} \cup \{\Omega\}$ *and* $t \in \mathcal{T}$*, by induction on the*

pair (d_Ω, t) *ordered lexicographically. The predicate is defined as follows:*

$$\begin{aligned}
(\bot : \bot) &= \mathsf{true} \\
(c : b) &= c \in \mathbb{B}(b) \\
((d_1, d_2) : t_1 \times t_2) &= (d_1 : t_1) \text{ and } (d_2 : t_2) \\
(\{(d^1, d^1_\Omega), \dots, (d^n, d^n_\Omega)\} : t_1 \to t_2) &= \forall i \in \{1, \dots, n\}.\ \mathsf{if}\ (d^i : t_1)\ \mathsf{then}\ (d^i_\Omega : t_2) \\
(d : t_1 \vee t_2) &= (d : t_1) \text{ or } (d : t_2) \\
(d : \neg t) &= \mathsf{not}\ (d : t) \\
(d_\Omega : t) &= \mathsf{false} \qquad \textit{otherwise}
\end{aligned}$$

We define the set-theoretic interpretation $[\![\cdot]\!] : \mathcal{T} \to \mathcal{P}(\mathcal{D})$ *as* $[\![t]\!] = \{\, d \in \mathcal{D} \mid (d : t) \,\}$.

Finally, we define the subtyping preorder and its associated equivalence relation as follows.

▶ **Definition 2.4** (Subtyping relation)**.** *We define the* subtyping *relation* $\leq$ *and the* subtyping equivalence *relation* $\simeq$ *as* $t_1 \leq t_2 \stackrel{\text{def}}{\iff} [\![t_1]\!] \subseteq [\![t_2]\!]$ *and* $t_1 \simeq t_2 \stackrel{\text{def}}{\iff} (t_1 \leq t_2) \text{ and } (t_2 \leq t_1)$.

3 Language syntax and semantics

We consider a language based on that studied in [20]: a λ-calculus with recursive explicitly annotated functions, pair constructors and destructors, and a typecase construct. This is the *source language* in which programs are written. We define the semantics on a slightly different *internal language* and show how to compile source programs to this internal language. The main reason for introducing the internal language is that, to describe call-by-need semantics in a small-step operational style, we need to add to the source language a let construct, a form of explicit substitution which models sharing of computations (following a standard approach [2,3,21]). The internal language is not an extension of the source language, however, because we also restrict the allowed syntax of typecases to simplify the semantics.

First, we give some auxiliary definitions on types. We introduce the abbreviations: $\langle t \rangle \stackrel{\text{def}}{=} t \vee \bot$; $t_1 \twoheadrightarrow t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \to \langle t_2 \rangle$; and $t_1 \otimes t_2 \stackrel{\text{def}}{=} \langle t_1 \rangle \times \langle t_2 \rangle$. These are compact notations for types including $\bot$. The first, $\langle t \rangle$, is an abbreviated way to write the type of possibly diverging expressions whose result has type t. The latter two are used in type annotations. The intent is that programmers never write $\bot$ explicitly. Rather, they use the $\twoheadrightarrow$ and $\otimes$ constructors instead of $\to$ and $\times$ so that $\bot$ is introduced implicitly. The $\to$ and $\times$ constructors are never written directly in program. We define the following restricted grammars of types

$$T ::= b \mid T \otimes T \mid T \twoheadrightarrow T \mid T \vee T \mid \neg T \mid \mathbb{0} \qquad \tau ::= b \mid \tau \otimes \tau \mid \mathbb{0} \to \mathbb{1} \mid \tau \vee \tau \mid \neg \tau \mid \mathbb{0}$$

both of which are interpreted coinductively, with the same restrictions of regularity and contractivity as in the definition of types. The types defined by these grammars are the only ones which appear in programs: neither includes $\bot$ explicitly.

In particular, functions are annotated with T types, where the $\otimes$ and $\twoheadrightarrow$ forms are used to ensure that every type below a constructor is of the form $t \vee \bot$.

Typecases, instead, check τ types. The only arrow type that can appear in them is $\mathbb{0} \to \mathbb{1}$, which is the top type of functions (every well-typed function has this type). This restriction means that typecases will not be able to test the types of functions, but only, at most, whether a value is a function or not. This restriction is not imposed in [20], and actually it could be lifted here without difficulty. We include it because the purpose of typecases in our language is, to some extent, the modelling of pattern matching, which cannot test the type of functions. Restricting typecases on arrow types also facilitates the extension of the system with polymorphism and type inference.

3.1 Source language

The *source language expressions* are the terms $\mathbf{e}$ produced inductively by the grammar

$$\begin{aligned}&\mathbf{e} ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e} \mid \mathbf{e}\,\mathbf{e} \mid (\mathbf{e},\mathbf{e}) \mid \pi_i\,\mathbf{e} \mid (x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}\\ &\mathcal{I} ::= \textstyle\bigwedge_{i\in I} T'_i \to T_i \qquad\qquad |I| > 0\end{aligned}$$

where f and x range over a set $\mathcal{X}$ of *expression variables*, c over the set $\mathcal{C}$ of constants, i in $\pi_i\,\mathbf{e}$ over $\{1,2\}$, and where τ in $(x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}$ is such that $\tau \not\simeq \mathbb{0}$ and $\tau \not\simeq \mathbb{1}$.

Source language expressions include variables, constants, λ-abstractions, applications, pairs constructors $(\mathbf{e},\mathbf{e})$ and destructors $\pi_1\,\mathbf{e}$ and $\pi_2\,\mathbf{e}$, plus the typecase $(x = \mathbf{e}) \in \tau\ ?\ \mathbf{e} : \mathbf{e}$.

A λ-abstraction $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$ is a possibly recursive function, with recursion parameter f and argument x, both of which are bound in the body; the function is explicitly annotated with its type $\mathcal{I}$, which is a finite intersection of types of the form $T' \to T$.

A typecase expression $(x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2$ has the following intended semantics: $\mathbf{e}_0$ is evaluated until it can be determined whether it has type τ or not, then the selected branch ($\mathbf{e}_1$ if the result of $\mathbf{e}_0$ has type τ, $\mathbf{e}_2$ if it has type $\neg\tau$: one of the two cases always occurs) is evaluated in an environment where x is bound to the result of $\mathbf{e}_0$. Actually, to simplify the presentation, we will give a non-deterministic semantics in which we allow to evaluate $\mathbf{e}_0$ more than what is needed to ascertain whether it has type τ.

In the syntax definition above we have restricted the types τ in typecases asking both $\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$. A typecase checking the type $\mathbb{1}$ is useless: since all expressions have type $\mathbb{1}$, it immediately reduces to its first branch. Likewise, a typecase checking the type $\mathbb{0}$ reduces directly to the second branch. Therefore, the two cases are uninteresting to consider. We forbid them because this allows us to give a simpler typing rule for typecases. Allowing them is just a matter of adding two (trivial) typing rules specific to these cases, as we show later.

As customary, we consider expressions up to renaming of bound variables. In $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$, f and x are bound in $\mathbf{e}$. In $(x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2$, x is bound in $\mathbf{e}_1$ and $\mathbf{e}_2$.

We do not provide mechanisms to define cyclic data structures. For example, we do not have a direct syntactic construct to define the infinitely nested pair $(1,(1,\dots))$. We can define it by writing a fixpoint operator (which can be typed in our system since types can be recursive) or by defining and applying a recursive function which constructs the pair. A general letrec construct as in [2] might be useful in practice (for efficiency or to provide greater sharing) but we omit it here since we are only concerned with typing.

3.2 Internal language

The *internal language expressions* are the terms e produced inductively by the grammar

$$\begin{aligned}&e ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, e \mid e\,e \mid (e,e) \mid \pi_i\,e \mid (x = \varepsilon) \in \tau\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e\\ &\varepsilon ::= x \mid c \mid \mu f\colon \mathcal{I}.\, \lambda x.\, e \mid (\varepsilon,\varepsilon)\end{aligned}$$

where metavariables and conventions are as in the source language. There are two differences with respect to the source language. One is the introduction of the construct $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, which is a binder used to model sharing of computations in call-by-need semantics (in $\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2$, x is bound in e_2). The other difference is that typecases cannot check arbitrary expressions, but only expressions of the restricted form given by ε. This restriction simplifies the semantics of typecases.

A source language expression $\mathbf{e}$ can be compiled to an internal language expression $\lceil \mathbf{e} \rceil$ as follows. Compilation is straightforward for all expressions apart from typecases:

$$\lceil x \rceil = x \qquad \lceil c \rceil = c \qquad \lceil \mu f : \mathcal{I}.\, \lambda x.\, \mathbf{e} \rceil = \mu f : \mathcal{I}.\, \lambda x.\, \lceil \mathbf{e} \rceil$$
$$\lceil \mathbf{e}_1\, \mathbf{e}_2 \rceil = \lceil \mathbf{e}_1 \rceil\, \lceil \mathbf{e}_2 \rceil \qquad \lceil (\mathbf{e}_1, \mathbf{e}_2) \rceil = (\lceil \mathbf{e}_1 \rceil, \lceil \mathbf{e}_2 \rceil) \qquad \lceil \pi_i\, \mathbf{e} \rceil = \pi_i\, \lceil \mathbf{e} \rceil$$

and for typecases it introduces a let binder to ensure that the checked expression is a variable:

$$\lceil (x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2 \rceil = \mathsf{let}\ y = \lceil \mathbf{e}_0 \rceil\ \mathsf{in}\ (x = y) \in \tau\ ?\ \lceil \mathbf{e}_1 \rceil : \lceil \mathbf{e}_2 \rceil$$

where y is chosen not free in $\mathbf{e}_1$ and $\mathbf{e}_2$. (The other forms for ε appear during reduction.)

3.3 Semantics

We define the operational semantics of the internal language as a small-step reduction relation using call-by-need. The semantics of the source language is then given indirectly through the translation. The choice of call-by-need rather than call-by-name was briefly motivated in the Introduction and will be discussed more extensively in Section 4.

We first define the sets of *answers* (ranged over by a) and of *values* (ranged over by v) as the subsets of expressions produced by the following grammars:

$$a ::= c \mid \mu f : \mathcal{I}.\, \lambda x.\, e \mid (e, e) \mid \mathsf{let}\ x = e\ \mathsf{in}\ a \qquad\qquad v ::= c \mid \mu f : \mathcal{I}.\, \lambda x.\, e$$

Answers are the results of evaluation. They correspond to expressions which are fully evaluated up to their top-level constructor (constant, function, or pair) but which may include arbitrary expressions below that constructor (so we have (e, e) rather than (a, a)). Since they also include let bindings, they represent closures in which variables can be bound to arbitrary expressions. Values are a subset of answers treated specially in a reduction rule.

The semantics uses evaluation contexts to direct the order of evaluation. A *context* C is an expression with a hole (written $[\,]$) in it. We write $C[e]$ for the expression obtained by replacing the hole in C with e. We write $C\lceil e \rceil$ for $C[e]$ when the free variables of e are not bound by C: for example, $\mathsf{let}\ x = e_1\ \mathsf{in}\ x$ is of the form $C[x]$ – with $C \equiv (\mathsf{let}\ x = e_1\ \mathsf{in}\ [\,])$ – but not of the form $C\lceil x \rceil$; conversely, $\mathsf{let}\ x = e_1\ \mathsf{in}\ y$ is both of the form $C[y]$ and $C\lceil y \rceil$.

Evaluation contexts E are the subset of contexts generated by the following grammar:

$$E ::= [\,] \mid E\, e \mid \pi_i\, E \mid (x = F) \in \tau\ ?\ e : e \mid \mathsf{let}\ x = e\ \mathsf{in}\ E \mid \mathsf{let}\ x = E\ \mathsf{in}\ E\lceil x \rceil$$
$$F ::= [\,] \mid (F, \varepsilon) \mid (\varepsilon, F)$$

Evaluation contexts allow reduction to occur on the left of applications and below projections, but not on the right of applications and below pairs. For typecases alone, the contexts allow reduction also below pairs, since this reduction might be necessary to be able to determine whether the expression has type τ or not. This is analogous to the behaviour of pattern matching in lazy languages, which can force evaluation below constructors. The contexts for let are from standard presentations of call-by-need [2, 21]. They allow reduction of the body of the let, while they only allow reductions of the bound expression when it is required to continue evaluating the body: this is enforced by requiring the body to have the form $E\lceil x \rceil$.

Figure 1 presents the reduction rules. They rely on the typeof function, which assigns types to expressions of the form ε. It is defined as follows:

$$\mathsf{typeof}(x) = \mathbb{1} \qquad \mathsf{typeof}(\mu f : \mathcal{I}.\, \lambda x.\, e) = \mathbb{0} \to \mathbb{1}$$
$$\mathsf{typeof}(c) = b_c \qquad \mathsf{typeof}((\varepsilon_1, \varepsilon_2)) = \mathsf{typeof}(\varepsilon_1) \times \mathsf{typeof}(\varepsilon_2)$$

$$
\begin{array}{rrcll}
[\textsc{Appl}] & (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\; e' & \rightsquigarrow & \mathsf{let}\ f = (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\ \mathsf{in}\ \mathsf{let}\ x = e'\ \mathsf{in}\ e & \\
[\textsc{ApplL}] & (\mathsf{let}\ x = e\ \mathsf{in}\ a)\; e' & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ a\; e' & \\
[\textsc{Proj}] & \pi_i\ (e_1, e_2) & \rightsquigarrow & e_i & \\
[\textsc{ProjL}] & \pi_i\ (\mathsf{let}\ x = e\ \mathsf{in}\ a) & \rightsquigarrow & \mathsf{let}\ x = e\ \mathsf{in}\ \pi_i\ a & \\
[\textsc{LetV}] & \mathsf{let}\ x = v\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & (E[\![x]\!])[v/x] & \\
[\textsc{LetP}] & \mathsf{let}\ x = (e_1, e_2)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ x_1 = e_1\ \mathsf{in}\ \mathsf{let}\ x_2 = e_2\ \mathsf{in}\ (E[\![x]\!])[(x_1,x_2)/x] & \\
[\textsc{LetL}] & \mathsf{let}\ x = (\mathsf{let}\ y = e\ \mathsf{in}\ a)\ \mathsf{in}\ E[\![x]\!] & \rightsquigarrow & \mathsf{let}\ y = e\ \mathsf{in}\ \mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!] & \\
[\textsc{Case1}] & (x = \varepsilon) \in \tau\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_1 & \text{if } \mathsf{typeof}(\varepsilon) \leq \tau \\
[\textsc{Case2}] & (x = \varepsilon) \in \tau\ ?\ e_1 : e_2 & \rightsquigarrow & \mathsf{let}\ x = \varepsilon\ \mathsf{in}\ e_2 & \text{if } \mathsf{typeof}(\varepsilon) \leq \neg\tau \\
[\textsc{Ctx}] & E[e] & \rightsquigarrow & E[e'] & \text{if } e \rightsquigarrow e'
\end{array}
$$

Figure 1 Operational semantics.

[Appl] is the standard application rule for call-by-need: the application $(\mu f\colon \mathcal{I}.\, \lambda x.\, e)\; e'$ reduces to e prefixed by two let bindings that bind the recursion variable f to the function itself and the parameter x to the argument e'. [ApplL] instead deals with applications with a let expression in function position: it moves the application below the let. The rule is necessary to prevent loss of sharing: substituting the binding of x to e in a would duplicate e. Symmetrically, there are two rules for pair projections, [Proj] and [ProjL].

There are three rules for let expressions. They rewrite expressions of the form $\mathsf{let}\ x = a\ \mathsf{in}\ E[\![x]\!]$: that is, let bindings where the bound expression is an answer and the body is an expression whose evaluation requires the evaluation of x. If a is a value v, [LetV] applies and the expression is reduced by just replacing v for x in the body. If a is a pair, [LetP] applies: the occurrences of x in the body are replaced with a pair of variables (x_1, x_2) and each x_i is bound to e_i by new let bindings (replacing x directly by (e_1, e_2) would duplicate expressions). Finally, the [LetL] rule just moves a let binding out of another.

There are two rules for typecases, by which a typecase construct $(x = \varepsilon) \in \tau\ ?\ e_1 : e_2$ can be reduced to either branch, introducing a new binding of x to ε. The rules apply only if either of $\mathsf{typeof}(\varepsilon) \leq \tau$ or $\mathsf{typeof}(\varepsilon) \leq \neg\tau$ holds. If neither holds, then the two rules do not apply, but the [Ctx] rule can be used to continue the evaluation of ε.

Comparison to other presentations of call-by-need. These reduction rules mirror those from standard presentations of call-by-need [2, 3, 21]. A difference is that, in [LetV] or [LetP], we replace *all* occurrences of x in $E[\![x]\!]$ at once, whereas in the cited presentations only the occurrence in the hole is replaced: for example, in [LetV] they reduce to $E[\![v]\!]$ instead of $(E[\![x]\!])[v/x]$. Our [LetV] rule is mentioned as a variant in [21, p. 38]. We use it because it simplifies the proof of subject reduction while maintaining an equivalent semantics.

Non-determinism in the rules. The semantics is not deterministic. There are two sources of non-determinism, both related to typecases. One is that the contexts F include both (F, ε) and (ε, F) and thereby impose no constraint on the order with which pairs are examined.

The second source of non-determinism is that the contexts for typecases allow us to reduce the bindings of variables in the checked expression even when we can already apply [Case1] or [Case2]. For example, take $\mathsf{let}\ x = e\ \mathsf{in}\ (y = (3, x)) \in (\mathsf{Int} \otimes \mathbb{1})\ ?\ e_1 : e_2$.

$$[\text{S-Subsum}]\ \frac{\Gamma \vdash \mathbf{e} \colon t' \qquad t' \leq t}{\Gamma \vdash \mathbf{e} \colon t} \qquad [\text{S-Var}]\ \frac{}{\Gamma \vdash x \colon t}\ \Gamma(x) = t \qquad [\text{S-Const}]\ \frac{}{\Gamma \vdash c \colon b_c}$$

$$[\text{S-Abstr}]\ \frac{\forall i \in I. \quad \Gamma, f \colon \mathcal{I}, x \colon \langle T_i' \rangle \vdash \mathbf{e} \colon \langle T_i \rangle}{\Gamma \vdash (\mu f \colon \mathcal{I}.\, \lambda x.\, \mathbf{e}) \colon \mathcal{I}}\ \mathcal{I} = \textstyle\bigwedge_{i \in I} T_i' \to T_i \qquad [\text{S-Appl}]\ \frac{\Gamma \vdash \mathbf{e}_1 \colon \langle t' \to t \rangle \qquad \Gamma \vdash \mathbf{e}_2 \colon t'}{\Gamma \vdash \mathbf{e}_1\, \mathbf{e}_2 \colon \langle t \rangle}$$

$$[\text{S-Pair}]\ \frac{\Gamma \vdash \mathbf{e}_1 \colon t_1 \qquad \Gamma \vdash \mathbf{e}_2 \colon t_2}{\Gamma \vdash (\mathbf{e}_1, \mathbf{e}_2) \colon t_1 \times t_2} \qquad [\text{S-Proj}]\ \frac{\Gamma \vdash \mathbf{e} \colon \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\, \mathbf{e} \colon \langle t_i \rangle}$$

$$[\text{S-Case}]\ \frac{\Gamma \vdash \mathbf{e}_0 \colon \langle t' \rangle \quad \big(\text{either } t' \leq \neg\tau \text{ or } \Gamma, x \colon (t' \wedge \tau) \vdash \mathbf{e}_1 \colon t\big) \quad \big(\text{either } t' \leq \tau \text{ or } \Gamma, x \colon (t' \setminus \tau) \vdash \mathbf{e}_2 \colon t\big)}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau\ ?\ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle}$$

Figure 2 Typing rules for the source language.

It can be immediately reduced to $\mathsf{let}\ x = e\ \mathsf{in}\ \mathsf{let}\ y = (3, x)\ \mathsf{in}\ e_1$ by applying [Ctx] and [Case1], because $\mathsf{typeof}((3, x)) = b_3 \times \mathbb{1} \leq \mathsf{Int} \otimes \mathbb{1}$. However, we can also use [Ctx] to reduce e, if it is reducible: we do so by writing the expression as $\mathsf{let}\ x = e\ \mathsf{in}\ E[x]$, where E is $(y = (3, [\,]))) \in (\mathsf{Int} \otimes \mathbb{1})\ ?\ e_1 : e_2$. To model a lazy implementation more faithfully, we should forbid this reduction and state that $(x = F) \in \tau\ ?\ e : e$ is a context only if it cannot be reduced by [Case1] or [Case2].

In both cases, we have chosen a non-deterministic semantics because it is less restrictive: as a consequence, the soundness result will also hold for semantics which fix an order.

4 Type system

We define two typing relations for the source language and the internal language.

A *type environment* Γ is a finite mapping of type variables to types. We write $\varnothing$ for the empty environment. We say that a type environment Γ is *well-formed* if, for all $(x \colon t) \in \Gamma$, we have $t \not\simeq \mathbb{0}$. Since we want to ensure that the empty type is never derivable, we will only consider well-formed type environments in the soundness proof.

4.1 Type system for the source language

Figure 2 presents the typing rules for the source language. The subsumption rule [S-Subsum] is used to apply subtyping. Notably, it allows expressions with surely converging types (like a pair with type $\mathsf{Int} \times \mathsf{Bool}$) to be used where diverging types are expected: $t \leq \langle t \rangle$ holds for every t (since $[\![t]\!] \subseteq [\![t]\!] \cup \{\bot\} = [\![t \vee \bot]\!] = [\![\langle t \rangle]\!]$). The rules [S-Var] and [S-Const] for variables and constants are standard. The [S-Abstr] rule for functions is also straightforward. Function interfaces have the form $\bigwedge_{i \in I} T_i' \to T_i$, that is, $\bigwedge_{i \in I} \langle T_i' \rangle \to \langle T_i \rangle$ (expanding the definition of $\to$). To type a function $\mu f \colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$, we check that it has all the arrow types in $\mathcal{I}$. Namely, for every arrow $T_i' \to T_i$ (i.e., $\langle T_i' \rangle \to \langle T_i \rangle$), we assume that x has type $\langle T_i' \rangle$ and that the recursion variable f has type $\mathcal{I}$, and we check that the body has type $\langle T_i \rangle$.

The [S-Appl] rule is the first one that deals with $\bot$ in a non-trivial way. In call-by-value semantic subtyping systems, to type an application $\mathbf{e}_1\, \mathbf{e}_2$ with a type t, the standard *modus ponens* rule (e.g., the one from the simply-typed λ-calculus) is used: $\mathbf{e}_1$ must have type $t' \to t$

and $\mathbf{e}_2$ must have type t'. Here, instead, we allow the function to have the type $\langle t' \to t \rangle$ (i.e., $(t' \to t) \vee \bot$) to make application possible also when $\mathbf{e}_1$ might diverge. We use $\langle t \rangle$ as the type of the whole application, signifying that it might diverge. As anticipated, we do not try to predict whether applications will converge. The rule [S-Pair] for pairs is standard; [S-Proj] handles $\bot$ as in applications.

[S-Case] is the most complex rule, but it corresponds closely to that of [20]. Strictly speaking it is not a single inference rule, but a shorthand way of writing four distinct rules with partially different premises and side conditions, here abbreviated in the form "either ... or ...". To type $(x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2$ we first type $\mathbf{e}_0$ with some type $\langle t' \rangle$. Then, we type the two branches $\mathbf{e}_1$ and $\mathbf{e}_2$. We do not always have to type both (because of the "either ... or ..." conditions) but for now assume that we do. While typing either branch, we extend the environment with a binding for x. For the first branch, the type for x is $t' \wedge \tau$, a subtype of $\langle t' \rangle$: this type is sound because the first branch is only evaluated if $\mathbf{e}_0$ evaluates to an answer (meaning we can remove the union with $\bot$ in $\langle t' \rangle$) and if this answer has type τ. Conversely, for the second branch, x is given type $t' \setminus \tau$, that is, $t' \wedge \neg\tau$. Finally, if the branches have type t, the whole typecase is given type $\langle t \rangle$ since its evaluation may diverge in case $\mathbf{e}_0$ diverges.

Now let us consider the conditions "either ... or ...". We need to type the first branch only when $t' \not\leq \neg\tau$; if, conversely, $t' \leq \neg\tau$, then we know that the first branch can never be selected (an expression of type $\neg\tau$ cannot reduce to a result of type τ) and thus we do not need to type it. The reasoning for the second branch is analogous. The two conditions are pivotal to type overloaded functions defined by typecases. For example, a negation function implemented as $\mu f\colon \mathcal{I}.\, \lambda x.\, (y = x) \in b_{\mathsf{true}} \ ? \ \mathsf{false} : \mathsf{true}$, with $\mathcal{I} = (b_{\mathsf{true}} \to b_{\mathsf{false}}) \wedge (b_{\mathsf{false}} \to b_{\mathsf{true}})$, could not be typed without these conditions.

In the syntax we have restricted the type τ in typecases requiring $\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$. Typecases where these conditions do not hold are uninteresting, since they do not actually check anything. The rule [S-Case] would be unsound for them because these typecases can reduce to one branch even if $\mathbf{e}_0$ is a diverging expression that does not evaluate to an answer. For instance, if $\bar{\mathbf{e}}$ has type $\bot$ (that is, $\langle \mathbb{0} \rangle$), then $(x = \bar{\mathbf{e}}) \in \mathsf{Int} \ ? \ 1 : 2$ could be given any type, including unsound ones like $\langle \mathsf{Bool} \rangle$. To allow these typecases, we could add the side condition "$\tau \not\simeq \mathbb{1}$ and $\tau \not\simeq \mathbb{0}$" to [S-Case] and give two specialized rules as follows:

$$\frac{\Gamma \vdash \mathbf{e}_0 \colon t' \qquad \Gamma, x \colon t' \vdash \mathbf{e}_1 \colon t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle} \ \tau \simeq \mathbb{1} \qquad \frac{\Gamma \vdash \mathbf{e}_0 \colon t' \qquad \Gamma, x \colon t' \vdash \mathbf{e}_2 \colon t}{\Gamma \vdash \big((x = \mathbf{e}_0) \in \tau \ ? \ \mathbf{e}_1 : \mathbf{e}_2\big) \colon \langle t \rangle} \ \tau \simeq \mathbb{0}$$

4.2 Type system for the internal language

Figure 3 presents the typing rules for the internal language. These include a new rule for let expressions and a modified rule for λ-abstractions; the other rules are the same as those for the source language (except for the different syntax of typecases).

The [S-Abstr] rule for the source language derived the type $\mathcal{I}$ for $\mu f\colon \mathcal{I}.\, \lambda x.\, \mathbf{e}$. The rule for the internal language, instead, allows us to derive a subtype of $\mathcal{I}$ of the form $\mathcal{I} \wedge t$, where t is an intersection of negations of arrow types. The arrows in t can be chosen freely providing that the intersection $\mathcal{I} \wedge t$ remains non-empty. This rule (directly taken from [20]) can look surprising. For example, it allows us to type $\mu f\colon (\mathsf{Int} \twoheadrightarrow \mathsf{Int}).\, \lambda x.\, x$ as $(\mathsf{Int} \twoheadrightarrow \mathsf{Int}) \wedge \neg(\mathsf{Bool} \to \mathsf{Bool})$ even though, disregarding the interface, the function does map booleans to booleans. But the language is explicitly typed, and thus we can't ignore interfaces (indeed, the function does not have type $\mathsf{Bool} \to \mathsf{Bool}$). The purpose of the rule is to ensure that, given any function and any type t, either the function has type t or it has type $\neg t$.

$$[\textsc{Subsum}]\ \frac{\Gamma \vdash e\colon t' \qquad t' \leq t}{\Gamma \vdash e\colon t} \qquad [\textsc{Var}]\ \frac{}{\Gamma \vdash x\colon t}\ \Gamma(x) = t \qquad [\textsc{Const}]\ \frac{}{\Gamma \vdash c\colon b_c}$$

$$[\textsc{Abstr}]\ \frac{\forall i \in I.\quad \Gamma, f\colon \mathcal{I}, x\colon \langle T'_i \rangle \vdash e\colon \langle T_i \rangle}{\Gamma \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon \mathcal{I} \wedge t} \quad \begin{array}{l} \mathcal{I} = \bigwedge_{i\in I} T'_i \to T_i \\ t = \bigwedge_{j\in J} \neg(t'_j \to t_j) \\ \mathcal{I} \wedge t \not\simeq \mathbb{0} \end{array} \qquad [\textsc{Appl}]\ \frac{\Gamma \vdash e_1\colon \langle t' \to t \rangle \qquad \Gamma \vdash e_2\colon t'}{\Gamma \vdash e_1\, e_2\colon \langle t \rangle}$$

$$[\textsc{Pair}]\ \frac{\Gamma \vdash e_1\colon t_1 \qquad \Gamma \vdash e_2\colon t_2}{\Gamma \vdash (e_1, e_2)\colon t_1 \times t_2} \qquad [\textsc{Proj}]\ \frac{\Gamma \vdash e\colon \langle t_1 \times t_2 \rangle}{\Gamma \vdash \pi_i\, e\colon \langle t_i \rangle}$$

$$[\textsc{Case}]\ \frac{\Gamma \vdash \varepsilon\colon \langle t' \rangle \quad \big(\text{either } t' \leq \neg\tau \text{ or } \Gamma, x\colon (t' \wedge \tau) \vdash e_1\colon t\big) \quad \big(\text{either } t' \leq \tau \text{ or } \Gamma, x\colon (t' \setminus \tau) \vdash e_2\colon t\big)}{\Gamma \vdash \big((x = \varepsilon) \in \tau\ ?\ e_1 : e_2\big)\colon \langle t \rangle}$$

$$[\textsc{Let}]\ \frac{\Gamma \vdash e_1\colon \bigvee_{i\in I} t_i \qquad \forall i \in I.\ \Gamma, x\colon t_i \vdash e_2\colon t}{\Gamma \vdash \mathsf{let}\ x = e_1\ \mathsf{in}\ e_2\colon t}$$

Figure 3 Typing rules for the internal language.

This property matches the intuitive view of types as sets of values that underpins semantic subtyping. While in our system we do not really interpret types as sets of values (since $\bot$ is non-empty and yet uninhabited by values), the property is still needed to prove subject reduction. A consequence of this property is that a value (i.e., a constant or a λ-abstraction) of type $t_1 \vee t_2$ has always either type t_1 or type t_2. (In the case of constants, this is obtained directly by reasoning on subtyping, so we don't need a rule to assign negation types to them.)

The [Let] rule combines a standard rule for (monomorphic) binders with a union disjunction rule: it lets us decompose the type of e_1 as a union and type the body of the let once for each summand in the union. The purpose of this rule was hinted at in the Introduction and will be discussed again in Section 4.3, where we show that this rule – combined with the property on union types above – is central to this work: it is the key technical feature that ensures the soundness of the system (see in particular Lemma 4.9 later on). For the time being, just note that the type of e_1 can be decomposed in arbitrarily complex ways by applying subsumption. For example, if e_1 is a pair of type $(\mathsf{Int} \vee \mathsf{Bool}) \times (\mathsf{Int} \vee \mathsf{Bool})$, by applying [Subsum] we can type it as $(\mathsf{Int} \times \mathsf{Int}) \vee (\mathsf{Int} \times \mathsf{Bool}) \vee (\mathsf{Bool} \times \mathsf{Int}) \vee (\mathsf{Bool} \times \mathsf{Bool})$ and then type e_2 once for each of the four summands.

The [Abstr] and [Let] rules introduce non-determinism in the choice of the negations to introduce and of how to decompose types as unions. This would not complicate a practical implementation, since a typechecker would only need to check the source language.

4.3 Properties of the type system

Full results about the type system, including proofs, are available in the extended version [22]. Here we report the main results and describe the technical difficulties we met to obtain them.

First, we can easily show by induction that compilation from the source language to the internal language preserves typing.

▶ **Proposition 4.1.** *If* $\Gamma \vdash \mathbf{e}\colon t$, *then* $\Gamma \vdash \lceil \mathbf{e} \rceil\colon t$.

We show the soundness property for our type system ("well-typed programs do not go wrong"), following the well-known syntactic approach of Wright and Felleisen [26], by proving the two properties of *progress* and *subject reduction* for the internal language.

▶ **Theorem 4.2** (Progress). *Let Γ be a well-formed type environment. Let e be an expression that is well-typed in Γ (that is, $\Gamma \vdash e : t$ holds for some t). Then e is an answer, or e is of the form $E[\![x]\!]$, or $\exists e'.\, e \rightsquigarrow e'$.*

▶ **Theorem 4.3** (Subject reduction). *Let Γ be a well-formed type environment. If $\Gamma \vdash e : t$ and $e \rightsquigarrow e'$, then $\Gamma \vdash e' : t$.*

The statement of progress is adapted to call-by-need: it applies also to expressions that are typed in a non-empty Γ and it allows a well-typed expression to have the form $E[\![x]\!]$.

As a corollary of these results, we obtain the following statement for soundness.

▶ **Corollary 4.4** (Type soundness). *Let e be a well-typed, closed expression (that is, $\varnothing \vdash e : t$ holds for some t). If $e \rightsquigarrow^* e'$ and e' cannot reduce, then e' is an answer and $\varnothing \vdash e' : t$.*

The soundness result for the internal language implies soundness for the source language.

▶ **Corollary 4.5** (Type soundness for the source language). *Let $\mathbf{e}$ be a well-typed, closed source language expression (that is, $\varnothing \vdash \mathbf{e} : t$ holds for some t). If $\lceil \mathbf{e} \rceil \rightsquigarrow^* e'$ and e' cannot reduce, then e' is an answer and $\varnothing \vdash e' : t$.*

We summarize here some of the crucial properties required to derive the results above. We also resume the discussion of the motivations behind our choice of call-by-need.

We introduced the $\bot$ type for diverging expressions because assigning the type $\mathbb{0}$ to any expression causes unsoundness. We must hence ensure that no expression can be assigned the type $\mathbb{0}$. In well-formed type environments, we can prove this easily by induction.

▶ **Lemma 4.6.** *Let Γ be a well-formed type environment. If $\Gamma \vdash e : t$, then $t \not\simeq \mathbb{0}$.*

Call-by-name and call-by-need. In the Introduction, we have given two reasons for our choice of call-by-need rather than call-by-name. One is that the system is only sound for call-by-name if we make assumptions on the semantics that might not hold in an extended language: for example, introducing an expression that can reduce non-deterministically either to an integer or to a boolean would break soundness. The other reason is that, even when these assumptions hold (and when presumably call-by-name and call-by-need are observationally equivalent), call-by-need is better suited to the soundness proof.

Let us review the example from the Introduction. Consider the function $\mu f : \mathcal{I}.\, \lambda x.\, (x, x)$ in the source language, where $\mathcal{I} = (\mathsf{Int} \twoheadrightarrow \mathsf{Int} \otimes \mathsf{Int}) \wedge (\mathsf{Bool} \twoheadrightarrow \mathsf{Bool} \otimes \mathsf{Bool})$. It is well-typed with type $\mathcal{I}$. By subsumption, it also has the type $(\mathsf{Int} \vee \mathsf{Bool}) \twoheadrightarrow (\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool})$, which is a supertype of $\mathcal{I}$: in general we have $(t_1' \to t_1) \wedge (t_2' \to t_2) \leq (t_1' \vee t_2') \to (t_1 \vee t_2)$ and therefore $(t_1' \twoheadrightarrow t_1) \wedge (t_2' \twoheadrightarrow t_2) \leq (t_1' \vee t_2') \twoheadrightarrow (t_1 \vee t_2)$.

Therefore, if $\bar{\mathbf{e}}$ has type $\mathsf{Int} \vee \mathsf{Bool} \vee \bot$, the application $(\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}$ is well-typed with type $(\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool}) \vee \bot$. Assume that $\bar{\mathbf{e}}$ can reduce either to an integer or to a boolean: for instance, assume that both $\bar{\mathbf{e}} \rightsquigarrow 3$ and $\bar{\mathbf{e}} \rightsquigarrow \mathsf{true}$ can occur.

With call-by-name, $(\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}$ reduces to $(\bar{\mathbf{e}}, \bar{\mathbf{e}})$; then, the two occurrences of $\bar{\mathbf{e}}$ reduce independently. It is intuitively unsound to type it as $(\mathsf{Int} \otimes \mathsf{Int}) \vee (\mathsf{Bool} \otimes \mathsf{Bool}) \vee \bot$: there is no guarantee that the two components of the pair will be of the same type once they are reduced. We can find terms that break subject reduction. Assume for example that there exists a boolean "and" operation; then this typecase is well-typed (as $\langle \mathsf{Bool} \rangle$) but unsafe:

$$(y = (\mu f : \mathcal{I}.\, \lambda x.\, (x, x))\, \bar{\mathbf{e}}) \in (\mathsf{Int} \otimes \mathsf{Int})\ ?\ \mathsf{true} : (\pi_1\, y\ \mathsf{and}\ \pi_2\, y)\ .$$

Since the application has type $\langle(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})\rangle$, to type the second branch of the typecase we can assume that y has the type $((\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool}))\setminus(\mathsf{Int}\otimes\mathsf{Int})$, which is a subtype of $\mathsf{Bool}\otimes\mathsf{Bool}$ (it is actually equivalent to $(\mathsf{Bool}\otimes\mathsf{Bool})\setminus(\bot\times\bot)$). Therefore, both $\pi_1\,y$ and $\pi_2\,y$ have type $\langle\mathsf{Bool}\rangle$. We deduce then that $(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)$ has type $\langle\mathsf{Bool}\rangle$ as well (we assume that "and" is defined so as to handle arguments of type $\bot$ correctly).

A possible reduction in a call-by-name semantics would be the following:

$$\begin{aligned}&(y=(\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x))\,\bar{\mathbf{e}})\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)\\ \rightsquigarrow\ &(y=(\bar{\mathbf{e}},\bar{\mathbf{e}}))\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)\end{aligned}$$

(the typecase must force the evaluation of $(\bar{\mathbf{e}},\bar{\mathbf{e}})$ to know which branch should be selected)

$$\rightsquigarrow^*\ (y=(\mathsf{true},\bar{\mathbf{e}}))\in(\mathsf{Int}\otimes\mathsf{Int})\ ?\ \mathsf{true}:(\pi_1\,y\ \mathsf{and}\ \pi_2\,y)$$

(now we know that the first branch is impossible, so the second is chosen)

$$\rightsquigarrow\ \pi_1\,(\mathsf{true},\bar{\mathbf{e}})\ \mathsf{and}\ \pi_2\,(\mathsf{true},\bar{\mathbf{e}})\ \rightsquigarrow\ \mathsf{true}\ \mathsf{and}\ \bar{\mathbf{e}}\ \rightsquigarrow\ \bar{\mathbf{e}}\ \rightsquigarrow\ 3$$

The integer 3 is not a Bool: this disproves subject reduction for call-by-name if the language contains expressions like $\bar{\mathbf{e}}$. No such expressions exist in our current language, but they could be introduced if we extended it with non-deterministic constructs like $\mathsf{rnd}(t)$ from [20].

Since we use a call-by-need semantics, instead, expressions such as $\bar{\mathbf{e}}$ do not pose problems for soundness. With call-by-need, $(\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x))\,\bar{\mathbf{e}}$ reduces to $\mathsf{let}\ f=\mu f\colon\mathcal{I}.\,\lambda x.\,(x,x)\ \mathsf{in}\ \mathsf{let}\ x=\bar{\mathbf{e}}\ \mathsf{in}\ (x,x)$. The occurrences of x in the pair are only substituted when $\bar{\mathbf{e}}$ has been reduced to an answer, so they cannot reduce independently.

To ensure subject reduction, we allow the rule for let bindings to split unions in the type of the bound term. This means that the following derivation is allowed.

$$\frac{\Gamma\vdash\bar{\mathbf{e}}\colon\mathsf{Int}\vee\mathsf{Bool}\qquad\Gamma,x\colon\mathsf{Int}\vdash(x,x)\colon\mathsf{Int}\otimes\mathsf{Int}\qquad\Gamma,x\colon\mathsf{Bool}\vdash(x,x)\colon\mathsf{Bool}\otimes\mathsf{Bool}}{\Gamma\vdash\mathsf{let}\ x=\bar{\mathbf{e}}\ \mathsf{in}\ (x,x)\colon(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})}$$

Proving subject reduction: main lemmas. While the typing rule for let bindings is simple to describe, proving subject reduction for the reduction rules [LetV] and [LetP] (those that actually perform substitutions) is challenging. For the reduction $\mathsf{let}\ x=v\ \mathsf{in}\ E[\![x]\!]\rightsquigarrow(E[\![x]\!])[v/x]$, we show the following results.

▶ **Lemma 4.7.** *Let v be a value that is well-typed in Γ (i.e., $\Gamma\vdash v\colon t'$ holds for some t'). Then, for every type t, we have either $\Gamma\vdash v\colon t$ or $\Gamma\vdash v\colon\neg t$.*

▶ **Corollary 4.8.** *If $\Gamma\vdash v\colon\bigvee_{i\in I}t_i$, then there exists an $i_0\in I$ such that $\Gamma\vdash v\colon t_{i_0}$.*

Consider for example the reduction $\mathsf{let}\ x=v\ \mathsf{in}\ (x,x)\rightsquigarrow(v,v)$. If v has type $\mathsf{Int}\vee\mathsf{Bool}$, then $\mathsf{let}\,x=v\ \mathsf{in}\ (x,x)$ has type $(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})$ as in the derivation above. Without this corollary, for (v,v) we could only derive the type $(\mathsf{Int}\vee\mathsf{Bool})\times(\mathsf{Int}\vee\mathsf{Bool})$, which is not a subtype of the type deduced for the redex. Applying the corollary, we deduce that v has either type Int or Bool; in both cases (v,v) can be given the type $(\mathsf{Int}\otimes\mathsf{Int})\vee(\mathsf{Bool}\otimes\mathsf{Bool})$.

These results are also needed in semantic subtyping for strict languages to prove subject reduction for applications. To ensure them, following [20], we have added in the type system for the internal language the possibility of typing functions with negations of arrow types.

The reduction $\mathsf{let}\ x=(e_1,e_2)\ \mathsf{in}\ E[\![x]\!]\rightsquigarrow\ \mathsf{let}\ x_1=e_1\ \mathsf{in}\ \mathsf{let}\ x_2=e_2\ \mathsf{in}\ (E[\![x]\!])[(x_1,x_2)/x]$, instead, is dealt with by the following lemma.

▶ **Lemma 4.9.** *If* $\Gamma \vdash (e_1, e_2) \colon \bigvee_{i\in I} t_i$*, then there exist two types* $\bigvee_{j\in J} t_j$ *and* $\bigvee_{k\in K} t_k$ *such that* $\Gamma \vdash e_1 \colon \bigvee_{j\in J} t_j$*,* $\Gamma \vdash e_2 \colon \bigvee_{k\in K} t_k$*, and* $\forall j \in J.\, \forall k \in K.\, \exists i \in I.\, t_j \times t_k \leq t_i$*.*

This is the result we need for the proof: $\mathsf{let}\, x = (e_1, e_2)\, \mathsf{in}\, E[\![x]\!]$ is typed by assigning a union type to (e_1, e_2) and then typing $E[\![x]\!]$ once for every t_i in the union, while the reduct $\mathsf{let}\, x_1 = e_1\, \mathsf{in}\, \mathsf{let}\, x_2 = e_2\, \mathsf{in}\, (E[\![x]\!])[^{(x_1,x_2)}/_x]$ must be typed by typing e_1 and e_2 with two union types and then typing the substituted expression with every product $t_j \times t_k$. Showing that each $t_j \times t_k$ is a subtype of a t_i ensures that the substituted expression is well-typed. The proof consists in recognizing that the union $\bigvee_{i\in I} t_i$ must be a decomposition into a union of some type $t_1 \times t_2$ and that therefore t_1 and t_2 can be decomposed separately into two unions.

These results rely on the distinction between types that contain $\bot$ and those that do not: they would not hold if we assumed that every type implicitly contained $\bot$. For instance, adding $\bot$ implicitly to any type would essentially mean interpreting products as $[\![t_1 \times t_2]\!] = ([\![t_1]\!] \cup \{\bot\}) \times ([\![t_2]\!] \cup \{\bot\})$ instead of $[\![t_1 \times t_2]\!] = [\![t_1]\!] \times [\![t_2]\!]$. This would make Lemma 4.9 fail. Its proof relies on being able to find, given any type t such that $t \leq \mathbb{1} \times \mathbb{1}$ (that is, a type whose set-theoretic interpretation consists entirely of pairs), a union type $\bigvee_{i\in I} t_i^1 \times t_i^2$ such that $t \simeq \bigvee_{i\in I} t_i^1 \times t_i^2$ (Lemma A.10 in the extended version [22]). This would not hold with the modified interpretation: for example, the type $(\mathsf{Int} \times \mathsf{Bool}) \setminus (\mathbb{0} \times \mathbb{0})$ is a subtype of $\mathbb{1} \times \mathbb{1}$ but cannot be expressed as a union of product types.

Despite some technical difficulties, call-by-need seems quite suited to the soundness proof. Hence, it would probably be best to use it for the proof even if we assumed explicitly that the language does not include problematic expressions like $\mathsf{rnd}(t)$. Soundness would then also hold for a call-by-name semantics that it is observationally equivalent to call-by-need.

5 A discussion on the interpretation of types

We have shown in the previous sections that a set-theoretic interpretation of types, adapted to take into account divergence (Definition 2.3), can be the basis for designing a sound type system for languages with lazy evaluation. In this section, we analyze the relation between such an interpretation and the expressions that are actually definable in the language.

Let us first recap some notions from [20]. The initial intuition which guides semantic subtyping is to see a type as the set of values of that type in the language we consider: for example, to see $\mathsf{Int} \to \mathsf{Bool}$ as the set of λ-abstractions of type $\mathsf{Int} \to \mathsf{Bool}$. However, we cannot directly define the interpretation of a type t as the set $\{\, v \mid \varnothing \vdash v \colon t \,\}$, because the typing relation $\varnothing \vdash v \colon t$ depends on the definition of subtyping, which depends in turn on the interpretation of types. Frisch, Castagna and Benzaken [20] avoid this circularity by giving an interpretation $[\![\cdot]\!]$ of types as subsets of an interpretation domain where finite relations replace λ-abstractions.

This interpretation (like ours except that there is no $\bot$) is used to define subtyping and the typing relation. Then, the following result is shown:

$$\forall t_1, t_2.\, [\![t_1]\!] \subseteq [\![t_2]\!] \iff [\![t_1]\!]_{\mathcal{V}} \subseteq [\![t_2]\!]_{\mathcal{V}} \qquad \text{where } [\![t]\!]_{\mathcal{V}} \stackrel{\text{def}}{=} \{\, v \mid \varnothing \vdash v \colon t \,\}$$

This result states that a type t_1 is a subtype of a type t_2 ($t_1 \leq t_2$, which is defined as $[\![t_1]\!] \subseteq [\![t_2]\!]$) if and only if every value v that can be assigned the type t_1 can also be assigned the type t_2. Showing the result above implies that, once the type system is defined, we can indeed reason on subtyping by reasoning on inclusion between sets of values.[1]

[1] The circularity is avoided since the typing relation in $\{\, v \mid \varnothing \vdash v \colon t \,\}$ is defined using $[\![\cdot]\!]$ and not $[\![\cdot]\!]_{\mathcal{V}}$.

This result is useful in practice, since, when typechecking fails because a subtyping judgment $t_1 \leq t_2$ does not hold, we know that there exists a value v such that $\varnothing \vdash v\colon t_1$ holds while $\varnothing \vdash v\colon t_2$ does not. This value v can be shown as a witness to the unsoundness of the program while reporting the error.[2] Moreover, at a more foundational level, the result nicely formalizes the intuition that types statically approximate computations, in the sense that a type t corresponds to the set of all possible values of expressions of type t.

In the following we discuss how an analogous result could hold with a non-strict semantics. First of all, clearly the correspondence cannot be between interpretations of types and sets of values as in [20], since then we would identify $\bot$ with $\mathbb{0}$. Hence we should consider, rather than values, sets of "results" of some kind, including (a representation of) divergence.

However, whichever notion of result we consider, it is hard to define an interpretation domain of types such that the desired correspondence holds, that is, such that a type t corresponds to the set of all possible results of expressions of type t. As the reader can expect, the key challenge is to provide an interpretation where an arrow type $t_1 \to t_2$ corresponds, as it seems sensible, to the set of λ-abstractions $\{\, (\mu f\colon \mathcal{I}.\, \lambda x.\, e) \mid \varnothing \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon t_1 \to t_2 \,\}$. For instance, our proposed definition of $[\![\cdot]\!]$ is sound with respect to this correspondence, but not complete, that is, not precise enough. We devote the rest of this section to explain why and to discuss the possibility of obtaining a complete definition. Consider the type $\mathsf{Int} \to \mathbb{0}$. By Definition 2.3, we have

$$\begin{aligned}[\![\mathsf{Int} \to \mathbb{0}]\!] &= \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in [\![\mathsf{Int}]\!] \implies d' \in [\![\mathbb{0}]\!] \,\} \\ &= \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \notin [\![\mathsf{Int}]\!] \,\}\end{aligned}$$

(since $[\![\mathbb{0}]\!] = \varnothing$, the implication can only be satisfied if $d \notin [\![\mathsf{Int}]\!]$). This type is not empty, therefore, if a result similar to that of [20] held, we would expect to be able to find a function $\mu f\colon \mathcal{I}.\, \lambda x.\, e$ such that $\varnothing \vdash (\mu f\colon \mathcal{I}.\, \lambda x.\, e)\colon \mathsf{Int} \to \mathbb{0}$. Alas, no such function can be defined in our language. This is easy to check: interfaces must include $\bot$ in the codomain of every arrow (since they use the $\to$ form), so no interface can be a subtype of $\mathsf{Int} \to \mathbb{0}$. Lifting this syntactic restriction to allow any arrow type in interfaces would not solve the problem: for a function to have type $\mathsf{Int} \to \mathbb{0}$, its body must have type $\mathbb{0}$, which is impossible, and indeed *must* be impossible for the system to be sound. It is therefore to be expected that $\mathsf{Int} \to \mathbb{0}$ is uninhabited in the language. This means that our current definition of $[\![\mathsf{Int} \to \mathbb{0}]\!]$ as a non-empty type is imprecise.

Changing $[\![\cdot]\!]$ to make the types of the form $t \to \mathbb{0}$ empty is easy, but it does not solve the problem in general. Using intersection types we can build more challenging examples: for instance, consider the type $(\mathsf{Int} \vee \mathsf{Bool} \to \mathsf{Int}) \wedge (\mathsf{Int} \vee \mathsf{String} \to \mathsf{Bool})$. While neither codomain is empty, and neither arrow should be empty, the whole intersection should: no function, when given an Int as argument, can return a result which is both an Int and a Bool.

In the call-by-value case, it makes sense to have $\mathsf{Int} \to \mathbb{0}$ and the intersection type above be non-empty, because they are inhabited by functions that diverge on integers. This is because divergence is not represented in the types (or, to put it differently, because it is represented by the type $\mathbb{0}$). A type like $t_1 \to t_2$ is interpreted as a specification of *partial correctness*: a function of this type, when given an argument in t_1, either diverges or returns a result in t_2. In our system, we have introduced a separate non-empty type for divergence. Hence, we should see a type as specifying *total* correctness, where divergence is allowed only for functions whose codomain includes $\bot$.

[2] In case of a type error, the CDuce compiler shows to the programmer a default value for the type $t_1 \setminus t_2$. Some heuristics are used to build a value in which only the part relevant to the error is detailed.

Let us look again at the current interpretation of arrow types.

$$[\![t_1 \to t_2]\!] = \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}$$

An arrow type is seen as a set of finite relations: we represent functions extensionally and approximate them with all their finite subsets. We use relations instead of functions to account for non-determinism. Within a relation, a pair (d, d') means that the function returns the output d' on the input d; a pair (d, Ω) that the function crashes on d; divergence is represented simply by the absence of a pair. In this way, as said above, a function diverging on some element of $[\![t_1]\!]$ could erroneously belong to the set even if $[\![t_2]\!]$ does not contain $\bot$.

To formalize the requirement of totality on the domain, we could modify the definition in this way:

$$[\![t_1 \to t_2]\!] = \{\, R \in \mathcal{P}_{\text{fin}}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \mathsf{dom}(R) \supseteq [\![t_1]\!] \text{ and } \forall (d, d') \in R.\, d \in [\![t_1]\!] \implies d' \in [\![t_2]\!] \,\}$$

(where $\mathsf{dom}(R) = \{\, d \mid \exists d' \in \mathcal{D}.\, (d, d') \in R \,\}$).

However, if we consider only finite relations as above, the definition makes no sense, since $[\![t_1]\!] \subseteq \mathsf{dom}(R)$ can hold only when $[\![t_1]\!]$ is finite, whereas types can have infinite interpretations. On the contrary, if we allowed relations to be infinite, then the set $\mathcal{D}$ would have to satisfy the equality $\mathcal{D} = \mathcal{C} \uplus (\mathcal{D} \times \mathcal{D}) \uplus \mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega)$ (where $\uplus$ denotes disjoint union), but no such set exists: the cardinality of $\mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega)$ is always strictly greater than that of $\mathcal{D}$.

Frisch, Castagna and Benzaken [20] point out this problem and use finite relations in the domain to avoid it. They motivate this choice with the observation that, while finite relations are not really appropriate to describe functions in a language (since these might have an infinite domain), they are suitable to describe types as far as subtyping is concerned. Indeed, we do not really care what the elements in the interpretation of a type are, but only how they are related to those in the interpretations of other types. It can be shown that

$$\forall t_1, t_1', t_2, t_2'.\ [\![t_1' \to t_1]\!] \subseteq [\![t_2' \to t_2]\!] \iff ([\![t_1']\!] \rightharpoonup [\![t_1]\!]) \subseteq ([\![t_2']\!] \rightharpoonup [\![t_2]\!])$$

where $X \rightharpoonup Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\mathcal{D} \times \mathcal{D}_\Omega) \mid \forall (d, d') \in R.\, d \in X \implies d' \in Y \,\}$ builds the set of possibly infinite relations. This can be generalized to more complex types:

$$\left[\!\!\left[\bigwedge_{i \in P} t_i' \to t_i\right]\!\!\right] \subseteq \left[\!\!\left[\bigvee_{i \in N} t_i' \to t_i\right]\!\!\right] \iff \bigcap_{i \in P} ([\![t_i']\!] \rightharpoonup [\![t_i]\!]) \subseteq \bigcup_{i \in N} ([\![t_i']\!] \rightharpoonup [\![t_i]\!])\,.$$

In [20], the authors argue that the restriction to finite relations does not compromise the precision of subtyping. For reasons of space we do not elaborate further on this, and we direct the interested reader to their work and the notions of *extensional interpretation* and of *model* therein.

Let us try to proceed analogously in our case: that is, find a new interpretation of types that matches the behaviour of possibly infinite relations that are total on their domain, while introducing an approximation to ensure that the domain is definable. The latter point means, notably, that functions must be represented as finite objects. The following definition of a *model* specifies the properties that such an interpretation should satisfy.

▶ **Definition 5.1** (Model). *A function $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ is a* model *if the following hold:*

- *the set $\overline{\mathcal{D}}$ satisfies $\overline{\mathcal{D}} = \{\bot\} \uplus \mathcal{C} \uplus (\overline{\mathcal{D}} \times \overline{\mathcal{D}}) \uplus \overline{\mathcal{D}}^{\mathsf{fun}}$ for some set $\overline{\mathcal{D}}^{\mathsf{fun}}$;*
- *for all b, t, t_1, and t_2,*

$$(\!|\bot|\!) = \{\bot\} \qquad (\!|b|\!) = \mathbb{B}(b) \qquad (\!|t_1 \times t_2|\!) = (\!|t_1|\!) \times (\!|t_2|\!) \qquad (\!|t_1 \to t_2|\!) \subseteq (\!|\mathbb{0} \to \mathbb{1}|\!) = \overline{\mathcal{D}}^{\mathsf{fun}}$$
$$(\!|t_1 \vee t_2|\!) = (\!|t_1|\!) \cup (\!|t_2|\!) \qquad (\!|\neg t|\!) = \overline{\mathcal{D}} \setminus (\!|t|\!) \qquad (\!|\mathbb{0}|\!) = \varnothing$$

- *for every finite, non-empty intersection* $\bigwedge_{i\in P} t'_i \to t_i$ *and every finite union* $\bigvee_{i\in N} t'_i \to t_i$,

$$(\!|\textstyle\bigwedge_{i\in P} t'_i \to t_i|\!) \subseteq (\!|\bigvee_{i\in N} t'_i \to t_i|\!) \iff \bigcap_{i\in P} \left((\!|t'_i|\!) \to (\!|t_i|\!)\right) \subseteq \bigcup_{i\in N} \left((\!|t'_i|\!) \to (\!|t_i|\!)\right)$$

where $X \to Y \stackrel{\text{def}}{=} \{\, R \in \mathcal{P}(\overline{\mathcal{D}} \times \overline{\mathcal{D}}) \mid \mathsf{dom}(R) \supseteq X \text{ and } \forall (d, d') \in R.\, d \in X \implies d' \in Y \,\}$.

We set three conditions for an interpretation of types $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ to be a model. The first constrains $\overline{\mathcal{D}}$ to have the same structure as $\mathcal{D}$, except that we do not fix the subset $\overline{\mathcal{D}}^{\mathsf{fun}}$ in which arrow types are interpreted. The second condition fixes the definition of $(\!|\cdot|\!)$ completely except for arrow types. The third condition ensures that subtyping on arrow types behaves as set containment between the sets of relations that are total on the domains of the arrow types.[3]

An interesting result is that, even though we do not know whether an interpretation of types which is a model can actually be found, we can compare a hypothetical model with the interpretation $[\![\cdot]\!]$ defined in Section 2. Indeed $[\![\cdot]\!]$ turns out to be a sound approximation of every model; that is, the subtyping relation $\leq$ defined in Definition 2.4 from $[\![\cdot]\!]$ is contained in every subtyping relation $\leq_{(\!|\cdot|\!)}$ defined from some model $(\!|\cdot|\!)$. We have proven that this holds for non-recursive types:

▶ **Proposition 5.2.** *Let* $(\!|\cdot|\!) : \mathcal{T} \to \mathcal{P}(\overline{\mathcal{D}})$ *be a model. Let* t_1 *and* t_2 *be two finite (i.e., non-recursive) types. If* $[\![t_1]\!] \subseteq [\![t_2]\!]$*, then* $(\!|t_1|\!) \subseteq (\!|t_2|\!)$.

We conjecture that the result holds for recursive types too, but this proof is left for future work.

Showing that $(\!|\cdot|\!)$ exists would be important to understand the connection between our types and the semantics. To use $(\!|\cdot|\!)$ to define subtyping for the use of a typechecker, though, we would also need to show that the resulting definition is decidable. Otherwise, $[\![\cdot]\!]$ would remain the definition used in a practical implementation since it is sound and decidable, though less precise.

6 Conclusion

We have shown how to adapt the framework of semantic subtyping [20] to languages with non-strict semantics. Our type system uses the subtyping relation from [20] unchanged (except for the addition of $\bot$), while the typing rules are reworked to avoid the pathological behaviour of semantic subtyping on empty types. Notably, typing rules for constructs like application and projection must handle $\bot$ explicitly. This ensures soundness for call-by-need.

This approach ensures that the subtyping relation still behaves set-theoretically: we can still see union, intersection, and negation in types as the corresponding operations on sets. We can still use intersection types to express overloading.

The type $\bot$ we introduce has no analogue in well-known type systems like the simply typed λ-calculus or Hindley-Milner typing. However, $\bot$ never appears explicitly in programs (it does not appear in types of the forms T and τ given at the beginning of Section 3). Hence, programmers do not need to use it and to consider the difference between terminating and non-terminating types while writing function interfaces or typecases. Still, sub-expressions of a program can have types with explicit $\bot$ (e.g., the type $\mathsf{Int} \vee \bot$). Such types are not expressible in the grammar of types visible to the programmer. Accordingly, error reporting

[3] We do not use the error element Ω in the definition of $X \to Y$, because the totality requirement makes it unnecessary: errors on a given input can be represented in a relation by the absence of a pair.

is required to be more elaborated, to avoid mentioning internal types that are unknown to the programmer.

A different approach to use semantic subtyping with non-strict languages would be to change the interpretation of types (and, as a result, the definition of subtyping) to avoid the pathological behaviour on $\mathbb{0}$, and then to use standard typing rules.

We have explored this alternative approach, but we have not found it promising. A modified subtyping relation loses important properties – especially results on the decomposition of product types – that we need to prove soundness via subject reduction. The approach we have adopted here is more suited to this technical work. However, a modified subtyping relation could yield an alternative type system for the source language, provided that we can relate it to the current system for the internal language.

We also plan to study more expressive typing rules that can track termination with some precision. For example, we could change the application rule so that it does not always introduce $\bot$. In function interfaces, some arrows could include $\bot$ and some could not: then, overloaded function types would express that a function behaves differently on terminating or diverging arguments. For example, the function $\lambda x.\, x + 1$ could have type $(\mathsf{Int} \to \mathsf{Int}) \wedge (\bot \to \bot)$, while $\lambda x.\, 3$ could have type $\mathbb{1} \to \mathsf{Int}$: the first diverges on diverging arguments, the other always terminates. It would be interesting for future work to explore forms of termination analysis to obtain greater precision. The difficulty is to ensure that the type $\mathbb{0}$ remains uninhabited and that all diverging expressions still have types that include $\bot$. This is trivial in the current system, but it is no longer straightforward with more refined typing rules.

A further direction for future work is to extend the language and the type system we have considered with more features. Notably, polymorphism, gradual typing, and record types are needed to be able to type effectively the Nix Expression Language, which was the starting inspiration for our work.

References

1 Davide Ancona and Andrea Corradi. Semantic subtyping for imperative object-oriented languages. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA 2016, pages 568–587. ACM, 2016. doi:10.1145/2983990.2983992.

2 Zena M. Ariola and Matthias Felleisen. The call-by-need lambda calculus. *Journal of Functional Programming*, 7(3):265–301, 1997.

3 Zena M. Ariola, John Maraist, Martin Odersky, Matthias Felleisen, and Philip Wadler. A call-by-need lambda calculus. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, pages 233–246. ACM, 1995. doi:10.1145/199448.199507.

4 Véronique Benzaken, Giuseppe Castagna, and Alain Frisch. ℂDuce: an XML-centric general-purpose language. In *Proceedings of the 8th ACM SIGPLAN International Conference on Functional Programming*, ICFP '03, pages 51–63. ACM, 2003. doi:10.1145/944705.944711.

5 Véronique Benzaken, Giuseppe Castagna, Kim Nguyen, and Jérôme Siméon. Static and dynamic semantics of NoSQL languages. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '13, pages 101–114. ACM, 2013. doi:10.1145/2429069.2429083.

6 Gavin M. Bierman, Andrew D. Gordon, Cătălin Hriţcu, and David Langworthy. Semantic Subtyping with an SMT Solver. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*, ICFP '10, pages 105–116. ACM, 2010.

7 Venanzio Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, Volume 1, Issue 2, 2005. doi:10.2168/LMCS-1(2:1)2005.

8 Giuseppe Castagna and Alain Frisch. A gentle introduction to semantic subtyping. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 198–199. ACM, 2005. doi:10.1145/1069774.1069793.

9 Giuseppe Castagna, Hyeonseung Im, Kim Nguyen, and Véronique Benzaken. A core calculus for XQuery 3.0. In *Programming Languages and Systems*, pages 232–256. Springer, 2015.

10 Giuseppe Castagna and Victor Lanvin. Gradual typing with union and intersection types. *Proceedings of the ACM on Programming Languages*, 1(ICFP):41:1–41:28, 2017. doi:10.1145/3110285.

11 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, and Pietro Abate. Polymorphic functions with set-theoretic types. Part 2: local type inference and type reconstruction. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 289–302. ACM, 2015. doi:10.1145/2676726.2676991.

12 Giuseppe Castagna, Kim Nguyen, Zhiwu Xu, Hyeonseung Im, Sergueï Lenglet, and Luca Padovani. Polymorphic functions with set-theoretic types. Part 1: syntax, semantics, and evaluation. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 5–17. ACM, 2014. doi:10.1145/2535838.2535840.

13 Giuseppe Castagna, Tommaso Petrucciani, and Kim Nguyen. Set-theoretic types for polymorphic variants. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 378–391. ACM, 2016. doi:10.1145/2951913.2951928.

14 Giuseppe Castagna and Zhiwu Xu. Set-theoretic foundation of parametric polymorphism and subtyping. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 94–106. ACM, 2011. doi:10.1145/2034773.2034788.

15 Robert L. Constable and Scott Fraser Smith. Partial objects in constructive type theory. In *IEEE Symposium on Logic in Computer Science (LICS)*, pages 183–193, 1987.

16 Ornela Dardha, Daniele Gorla, and Daniele Varacca. Semantic subtyping for objects and classes. In *Formal Techniques for Distributed Systems*, pages 66–82. Springer, 2013.

17 Eelco Dolstra and Andres Löh. NixOS: a purely functional Linux distribution. In *Proceedings of the 13th ACM SIGPLAN International Conference on Functional Programming*, ICFP '08, pages 367–378. ACM, 2008. doi:10.1145/1411204.1411255.

18 Joshua Dunfield. *A unified system of type refinements.* PhD thesis, Carnegie Mellon University, 2007.

19 Joshua Dunfield and Frank Pfenning. Type assignment for intersections and unions in call-by-value languages. In *Foundations of Software Science and Computation Structures*, pages 250–266. Springer, 2003.

20 Alain Frisch, Giuseppe Castagna, and Véronique Benzaken. Semantic subtyping: dealing set-theoretically with function, union, intersection, and negation types. *Journal of the ACM*, 55(4):19:1–19:64, 2008. doi:10.1145/1391289.1391293.

21 John Maraist, Martin Odersky, and Philip Wadler. The call-by-need lambda calculus. *Journal of Functional Programming*, 8(3):275–317, 1998.

22 T. Petrucciani, G. Castagna, D. Ancona, and E. Zucca. Semantic subtyping for non-strict languages. Extended version. https://arxiv.org/abs/1810.05555, 2018.

23 Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282. ACM, 2014.

24 Jérôme Vouillon. Subtyping Union Types. In Jerzy Marcinkowski and Andrzej Tarlecki, editors, *Computer Science Logic*, pages 415–429. Springer, 2004.

25 Jérôme Vouillon and Paul-André Melliès. Semantic Types: A Fresh Look at the Ideal Model for Types. In *Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '04, pages 52–63. ACM, 2004. doi:10.1145/964001.964006.

26 Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994. doi:10.1006/inco.1994.1093.

27 Francesco Zappa Nardelli, Julia Belyakova, Artem Pelenitsyn, Benjamin Chung, Jeff Bezanson, and Jan Vitek. Julia subtyping: A rational reconstruction. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):113:1–113:27, 2018. doi:10.1145/3276483.

New Formalized Results on the Meta-Theory of a Paraconsistent Logic

Anders Schlichtkrull
DTU Compute - Department of Applied Mathematics and Computer Science,
Technical University of Denmark, Richard Petersens Plads, Building 324,
DK-2800 Kongens Lyngby, Denmark
andschl@dtu.dk

Abstract

Classical logics are explosive, meaning that everything follows from a contradiction. Paraconsistent logics are logics that are not explosive. This paper presents the meta-theory of a paraconsistent infinite-valued logic, in particular new results showing that while the question of validity for a given formula can be reduced to a consideration of only finitely many truth values, this does not mean that the logic collapses to a finite-valued logic. All definitions and theorems are formalized in the Isabelle/HOL proof assistant.

2012 ACM Subject Classification Theory of computation → Logic; Theory of computation → Logic and verification; Theory of computation → Higher order logic; Theory of computation → Logic and databases

Keywords and phrases Paraconsistent logic, Many-valued logic, Formalization, Isabelle proof assistant, Paraconsistency

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.5

Related Version An earlier version of the present paper appears as a chapter in my PhD thesis (`http://matryoshka.gforge.inria.fr/pubs/schlichtkrull_phd_thesis.pdf`) [15].

Acknowledgements Jørgen Villadsen, Jasmin Christian Blanchette and John Bruntse Larsen provided valuable feedback on the paper and the formalization. Thanks to Freek Wiedijk for discussions. Thanks to the anonymous reviewers for many constructive comments.

1 Introduction

Classical logics are by design *explosive* – everything follows from a contradiction. This is mostly uncontroversial, but it seems problematic for certain kinds of reasoning. In paraconsistent logics, everything does not follow from a contradiction. Non-classical logics should also enjoy the benefits of formalization, and therefore this paper presents a formalization of a paraconsistent infinite-valued propositional logic.

The entry on paraconsistent logic in the Stanford Encyclopedia of Philosophy [13] thoroughly motivates paraconsistent logics by arguing that some domains do contain inconsistencies, but this should not make meaningful reasoning impossible. An example from computer science is that in large knowledge bases an inconsistency can easily occur if just one data point is entered wrong. A reasoning system based on such a database needs a meaningful way to deal with the inconsistency. Many other examples are mentioned from philosophy, linguistics, automated reasoning and mathematics. A recent book [1] looks at paraconsistency in the domain of engineering. There is no one paraconsistent logic to rule them all – there are many logics which can be used in different contexts. The encyclopedia gives a taxonomy of paraconsistent logics consisting of discussive logics, non-adjunctive systems, preservationism, adaptive logics, logics of formal inconsistency, relevant logics and many-valued logics.

Table 1 This table shows where the results of this paper have been conjectured and where their formal and informal proofs have previously been presented.

Results in	Conjecture	Formal proof	Informal proof
Section 4	Jensen and Villadsen [10]	Villadsen and me [23, 24]	the present paper
Section 5	Jensen and Villadsen [10]	the present paper	the present paper
Section 6	Villadsen and me [23, 24]	the present paper	the present paper

The logic considered here is the propositional fragment of a paraconsistent infinite-valued higher-order logic by Villadsen [19, 21, 20, 22] and more recently Jensen and Villadsen in an extended abstract [10]. The propositional logic, here called $\mathbb{V}$, has a semantics with the two classical truth values and countably infinitely many non-classical truth values. When U is a subset of the non-classical truth values, $\mathbb{V}_U$ is the logic defined the same as $\mathbb{V}$ except for the restriction that its non-classical truth values are only those in U. This does not require any change of the semantics of $\mathbb{V}$'s logical operators because they are defined in a way such that when their domain is restricted then their range is similarly restricted. In $\mathbb{V}_U$ with a finite U, one can find out whether a formula p is valid by enumerating enough interpretations that they cover all possible assignments of the propositional symbols in p. This approach does not work in $\mathbb{V}$ since there are infinitely many such interpretations. This paper shows that it is enough to consider the models in $\mathbb{V}_U$ for a finite U, but that the size of U depends on the formula considered.

The contents of this paper are as follows:

- Section 2 defines and formalizes $\mathbb{V}$. It gives an example of paraconsistency in the logic.
- Section 3 defines and formalizes $\mathbb{V}_U$.
- Section 4 proves and formalizes that for any formula p there is a finite U such that if p is valid in $\mathbb{V}_U$, it is also valid in $\mathbb{V}$. This allows the question of validity in $\mathbb{V}$ to be solved by a finite enumeration of interpretations.
- Section 5 proves and formalizes the new result that if $|U| = |W|$, then $\mathbb{V}_U$ and $\mathbb{V}_W$ consider the same formulas valid.
- Section 6 shows the new result that, to answer the question of validity in $\mathbb{V}$, one cannot fix a finite valued $\mathbb{V}_U$ once and for all because there exists a formula $\pi_{|U|}$ that is valid in this logic but not in $\mathbb{V}$. In other words, despite the result in Section 4, $\mathbb{V}$ is a different logic than any finite valued $\mathbb{V}_U$.

The formalization in Sections 2 and 3 was previously presented in a book chapter [23] and a paper [24] by Villadsen and myself. The result in Section 4 had already been conjectured by Jensen and Villadsen [10], but was, to the best of my knowledge, first proved and formalized in the mentioned book chapter [23] and paper [24]. The results in Section 5 were also conjectured by Jensen and Villadsen [10], but the results are, to the best of my knowledge, proved and formalized in the present paper for the first time. The result in Section 6 was conjectured by Villadsen and myself [24] and is, to the best of my knowledge, proved and formalized in the present paper for the first time except for a brief mention in the abstract of a talk by me [14]. For a summary of the appearances of the results see Table 1. Thus, there are no previous informal proofs to refer to for these results, and this paper will therefore both present the formalization of these results and their informal proofs. The full formalization is available online – 1500 lines of code are already in an *Archive of Formal Proofs* entry by Villadsen and myself [17], and the 800 lines corresponding to Sections 5 and 6 [16] will be added later. To make the paper easier to read, its notation is slightly different from the formalization.

2 A Paraconsistent Infinite-Valued Logic

The paraconsistent infinite-valued propositional logic $\mathbb{V}$ has two classical truth values, namely true ($\bullet$) and false ($\circ$). These are called the determinate truth values. True ($\bullet$) is the only designated value. The logic also has countably many different non-classical truth values ($_{\mathrm{I}}$, $_{\mathrm{II}}$, $_{\mathrm{III}}$, ...) [10]. These are called the indeterminate truth values. This is represented as a datatype *tv*.

datatype *tv* = *Det bool* | *Indet nat*

Det True and *Det False* represent $\bullet$ and $\circ$ respectively, and constructor *Indet* maps each natural number (0, 1, 2, ...) to the corresponding indeterminate truth value ($_{\mathrm{I}}$, $_{\mathrm{II}}$, $_{\mathrm{III}}$, ...).

The propositional symbols of $\mathbb{V}$ are strings of a finite alphabet. Here, the symbols are denoted as $\mathsf{p}, \mathsf{q}, \mathsf{r}, \ldots$. Interpretations are functions from propositional symbols into truth values. The formulas of the logic are built from the propositional symbols and operators $\neg$, $\wedge$, $\Leftrightarrow$ and $\leftrightarrow$ as well as a symbol for truth $\top$. To make them distinguishable, the logical operators in the paraconsistent logic are bold, while Isabelle/HOL's logical operators are not (e.g. $\neg$, $\wedge$, $\vee$, $\longleftrightarrow$). $\Leftrightarrow$ represents equality whereas $\neg$, $\wedge$ and $\leftrightarrow$ are designed to be generalizations of their classical counterparts. In the Isabelle/HOL formalization, the formulas are defined by a datatype *fm*, with a constructor for atomic formulas consisting of propositional symbols and with constructors for each of the operators. Additionally, a number of derived operators are defined:

$$\begin{array}{ll}
\bot \equiv \neg\, \top & \neg\!\!\neg\, p \equiv \Box\, (\neg\, p) \\
p \vee q \equiv \neg\, (\neg\, p \wedge \neg\, q) & p \barwedge q \equiv \Box\, (p \wedge q) \\
p \Rightarrow q \equiv p \Leftrightarrow (p \wedge q) & p \barvee q \equiv \Box\, (p \vee q) \\
p \rightarrow q \equiv p \leftrightarrow (p \wedge q) & \boldsymbol{\Delta}\ p \equiv (\Box\ p) \barvee (p \Leftrightarrow \bot) \\
\Box\ p \equiv p \Leftrightarrow \top & \boldsymbol{\nabla}\ p \equiv \neg\!\!\neg\, (\boldsymbol{\Delta}\ p)
\end{array}$$

In the semantics, Villadsen motivated the different cases by equalities of classical logic that also hold in $\mathbb{V}$ [19]. These motivating equalities are shown to the right of their case:

$$\begin{array}{l}
\mathit{eval}\ i\ x\ =\ i\ x \text{ if } x \text{ is a propositional symbol} \\
\mathit{eval}\ i\ \top\ =\ \bullet
\end{array}$$

$$\mathit{eval}\ i\ (\neg\, p)\ = \begin{cases}
\bullet & \text{if } \mathit{eval}\ i\ p = \circ \qquad \top \Leftrightarrow \neg\, \bot \\
\circ & \text{if } \mathit{eval}\ i\ p = \bullet \qquad \bot \Leftrightarrow \neg\, \top \\
\mathit{eval}\ i\ p & \text{otherwise}
\end{cases}$$

$$\mathit{eval}\ i\ (p \wedge q)\ = \begin{cases}
\mathit{eval}\ i\ p & \text{if } \mathit{eval}\ i\ p = \mathit{eval}\ i\ q \qquad p \Leftrightarrow p \wedge p \\
\mathit{eval}\ i\ q & \text{if } \mathit{eval}\ i\ p = \bullet \qquad q \Leftrightarrow \top \wedge q \\
\mathit{eval}\ i\ p & \text{if } \mathit{eval}\ i\ q = \bullet \qquad p \Leftrightarrow p \wedge \top \\
\circ & \text{otherwise}
\end{cases}$$

$$eval\ i\ (p \Leftrightarrow q) \;=\; \begin{cases} \bullet & \text{if } eval\ i\ p = eval\ i\ q \\ \circ & \text{otherwise} \end{cases}$$

$$eval\ i\ (p \leftrightarrow q) \;=\; \begin{cases} \bullet & \text{if } eval\ i\ p = eval\ i\ q & \top \;\Leftrightarrow\; p \leftrightarrow p \\ eval\ i\ q & \text{if } eval\ i\ p = \bullet & q \;\Leftrightarrow\; \top \leftrightarrow q \\ eval\ i\ p & \text{if } eval\ i\ q = \bullet & p \;\Leftrightarrow\; p \leftrightarrow \top \\ eval\ i\ (\neg\ q) & \text{if } eval\ i\ p = \circ & \neg q \;\Leftrightarrow\; \bot \leftrightarrow q \\ eval\ i\ (\neg\ p) & \text{if } eval\ i\ q = \circ & \neg\ p \;\Leftrightarrow\; p \leftrightarrow \bot \\ \circ & \text{otherwise} \end{cases}$$

Among the derived operators $\Box$, $\mathbf{\Delta}$ and $\mathbf{\nabla}$ are of special interest. $\Box$ maps $\bullet$ to $\bullet$ and any other value to $\circ$. In other words $\Box\ p$ states "p is true". Similarly $\mathbf{\Delta}\ p$ states "p is determinate" and $\mathbf{\nabla}\ p$ states "p is indeterminate".

The other operators can be divided in two groups – general operators ($\neg$, $\wedge$, $\vee$ and $\leftrightarrow$) and purely determinate operators ($\neg\!\!\neg$, $\curlywedge$, $\curlyvee$ and $\Leftrightarrow$). The general operators behave as expected on determinate values, and this behavior is generalized to indeterminate values. Consider for example the truth table in $\mathbb{V}_{\{\mathrm{I},\mathrm{II}\}}$ for $\vee$:

$\vee$	$\bullet$	$\circ$	I	II
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\circ$	$\bullet$	$\circ$	I	II
I	$\bullet$	I	I	$\bullet$
II	$\bullet$	II	$\bullet$	II

The purely determinate operators also behave as expected on determinate values, and their behavior generalizes to indeterminate values – however this time in such a way that they always return a determinate truth value. Consider for example the truth table in $\mathbb{V}_{\{\mathrm{I},\mathrm{II}\}}$ for $\curlyvee$:

$\curlyvee$	$\bullet$	$\circ$	I	II
$\bullet$	$\bullet$	$\bullet$	$\bullet$	$\bullet$
$\circ$	$\bullet$	$\circ$	$\circ$	$\circ$
I	$\bullet$	$\circ$	$\circ$	$\bullet$
II	$\bullet$	$\circ$	$\bullet$	$\circ$

Validity is defined in the usual way, i.e. a formula is valid if it is true in all interpretations.

definition *valid* :: *"fm ⇒ bool"*
where
"valid p ≡ ∀ i. eval i p = •"

Weber [25] explains that the literature contains two competing views on paraconsistency. One states that a logic is paraconsistent iff some formulas p and q exists such that $p, \neg\ p \nvdash q$. Another view states that a logic is paraconsistent iff some formulas p and q exist such that

$\vdash p$, $\vdash \neg\, p$ and $\nvdash q$. The logic $\mathbb{V}$ is paraconsistent with respect to the first of these views. Note that with this definition, paraconsistency is a property of entailment. Villadsen [21, 19] instead encodes this as the non-validity of a formula $(\mathsf{p} \wedge (\neg\, \mathsf{p})) \Rrightarrow \mathsf{q}$. This formula is not valid in $\mathbb{V}$ since it has e.g. the counter-model mapping p to $\shortmid$ and q to $\circ$. If one insists on a notion of entailment it can, for finite sets of formulas, simply be introduced by defining that $p_1, \ldots, p_n \vdash q$ iff $p_1 \wedge \ldots \wedge p_n \Rrightarrow q$ is valid [21, 20]. With this definition it follows that $\mathbb{V}$ is paraconsistent because then the above non-validity implies that there exist formulas p and q such that $\mathsf{p}, \neg\, \mathsf{p} \nvdash \mathsf{q}$.

3 Paraconsistent Finite-Valued Logics

For any set U of indeterminate truth values, the logic $\mathbb{V}_U$ is defined as follows: $\mathbb{V}_U$ is defined in the same way as $\mathbb{V}$, except that it has a different notion of interpretations. An interpretation in $\mathbb{V}_U$ is a function from propositional symbols to the set $\{\bullet, \circ\} \cup U$ instead of to the type of all truth values.

A function *domain* constructs $\{\bullet, \circ\} \cup U$ from a set of natural numbers:

definition *domain* :: "*nat set* ⇒ *tv set*"
where
"*domain U* ≡ {*Det True*, *Det False*} ∪ *Indet* ' *U*"

Here, *Indet* '*U* denotes the image of *Indet* on U. Notice that in the formalization, U is a set of natural numbers rather than a set of indeterminate values. This is only because it is less tedious to write $\{0, 1, 2\}$ than $\{\mathit{Indet}\ 0, \mathit{Indet}\ 1, \mathit{Indet}\ 2\}$ and because being able to write *domain* $\{\mathit{Indet}\ 0, \bullet\}$ is rather pointless since $\bullet$ is added by *domain* anyway. For the same reasons, I will from now on also write e.g. $\mathbb{V}_{\{0,1,2\}}$ rather than $\mathbb{V}_{\{\mathit{Indet}\ 0,\ \mathit{Indet}\ 1,\ \mathit{Indet}\ 2\}}$. The function is called *domain* because in the higher-order version of $\mathbb{V}$ one can use the truth values as the domain of discourse.

The notion of being valid in $\mathbb{V}_U$ is formalized. The expression *range i* denotes the function range of i.

definition *valid_in* :: "*nat set* ⇒ *fm* ⇒ *bool*"
where
"*valid_in U p* ≡ ∀ *i*. *range i* ⊆ *domain U* ⟶ *eval i p* = •"

It is clear that validity in $\mathbb{V}$ implies validity in any $\mathbb{V}_U$.

theorem *valid_valid_in*: **assumes** "*valid p*" **shows** "*valid_in U p*"

Proof. If p is valid in $\mathbb{V}$, it is true in all interpretations and thus in particular those with the desired range. Therefore p is valid in $\mathbb{V}_U$. ◀

The set U can be finite or infinite. The former case in particular will be of interest in the following sections.

4 A Reduction from Validity in $\mathbb{V}$ to Validity in $\mathbb{V}_U$

When U is finite, one can find out if a formula is valid by considering all the different cases of what an interpretation might map the formula's propositional symbols to. As an example, consider the formula $(\mathsf{p} \wedge (\neg\, \mathsf{p})) \longrightarrow \mathsf{q}$ in the logic $\mathbb{V}_\emptyset$, which corresponds to classical propositional logic.

```
proposition "valid_in ∅ ((p ∧ (¬ p)) → q)"
  unfolding valid_in_def
proof (rule, rule)
  fix i :: "id ⇒ tv"
  assume "range i ⊆ domain ∅"
  then have
      "i p ∈ {•, ○}"
      "i q ∈ {•, ○}"
    unfolding domain_def
    by auto
  then show "eval i ((p ∧ (¬ p)) → q) = •"
    by (cases "i p"; cases "i q") auto
qed
```

For $\mathbb{V}$ this approach does not work, since there are infinitely many truth values. This section overcomes the problem by showing that there exists a finite subset of the interpretations in $\mathbb{V}_U$ that it is enough to enumerate. The idea is that looking at the semantics of $\mathbb{V}$ reveals that there is a lot of symmetry between the indeterminate truth values $\mathrm{I}, \mathrm{II}, \mathrm{III}, \ldots$ Specifically, the indeterminate values are all different and can be told apart using $\Leftrightarrow$, but none of them play any special role compared with the others. Intuitively, this means that one just needs to consider enough interpretations to ensure that one has considered all different possibilities of interpreting the different pairs of propositional symbols as either different or equal indeterminate truth values. Therefore it is only necessary to consider enough truth values to ensure that this is possible and thus, for any formula p, it should be sufficient to consider all the interpretations in the logic $\mathbb{V}_U$, where $|U|$ is at least the number of propositional symbols in p.

The first step towards proving this is to prove that interpretations that agree on the propositional symbols occurring in a formula also evaluate the formula to the same result. The set of propositional symbols occurring is defined recursively by the following equations:

$$
\begin{aligned}
&\mathit{props}\ \top = \{\} \\
&\mathit{props}\ x = \{x\} \text{ if } x \text{ is a propositional symbol} \\
&\mathit{props}\ (\neg\ p) = \mathit{props}\ p \\
&\mathit{props}\ (p \wedge q) = \mathit{props}\ p \cup \mathit{props}\ q \\
&\mathit{props}\ (p \Leftrightarrow q) = \mathit{props}\ p \cup \mathit{props}\ q \\
&\mathit{props}\ (p \leftrightarrow q) = \mathit{props}\ p \cup \mathit{props}\ q
\end{aligned}
$$

Hereafter, the mentioned property is proved:

```
lemma relevant_props: assumes "∀ s ∈ props p. i₁ s = i₂ s" shows "eval i₁ p = eval i₂ p"
```

Proof. Follows by induction on the formula and the definitions of *props* and *eval*. ◀

The next step is to consider an interpretation i in $\mathbb{V}$ and see that it behaves the same as a corresponding interpretation in $\mathbb{V}_U$. The idea is that i can be changed to an interpretation in $\mathbb{V}_U$ by applying a function from *nat* into U to the indeterminate values that the interpretation returns.

Given a function f of type $\mathit{nat} \Rightarrow \mathit{nat}$ and an interpretation, its application $f\ x$ to a truth value x is defined as

$$
f\ x = \begin{cases} x & \text{if } x \text{ is determinate} \\ \mathit{Indet}\ (f\ n) & \text{if } x = \mathit{Indet}\ n \end{cases}
$$

A function can also be applied to an interpretation:

$f\ i = \lambda s.\ f\ (i\ s)$

If f is an injection, then applying f to the result or to the interpretation gives the same result when evaluating a formula.

lemma *eval_change*: **assumes** "*inj f*" **shows** "*eval* (*f i*) *p* = *f* (*eval i p*)"

Proof. The proof is by induction on the formula. In each inductive case the formula consists of one of the (non-derived) logical constructors and a number of immediate subformulas. Now look at how the semantics for that logical constructor was defined. For each operator, consider the different cases of what the subformulas could evaluate to under i as specified in the semantics. Doing this generates all in all 17 different cases. Consider for instance the semantics' "otherwise"-case for $p \leftrightarrow q$. Here, it is the case that $eval\ i\ p \neq eval\ i\ q$ and that there exists a natural number n such that $eval\ i\ p = Indet\ n$ and some m such that $eval\ i\ q = Indet\ m$. Hence $Indet\ n \neq Indet\ m$ and therefore $n \neq m$. Since f is injective, also $f\ n \neq f\ m$ and $Indet\ (f\ n) \neq Indet\ (f\ m)$. The induction hypotheses are $eval\ (f\ i)\ p = f\ (eval\ i\ p)$ and $eval\ (f\ i)\ q = f\ (eval\ i\ q)$. Consider the first one. Here it is the case that $eval\ (f\ i)\ p = f\ (eval\ i\ p) = f\ (Indet\ n) = Indet\ (f\ n)$. Likewise from the second it follows that $eval\ (f\ i)\ q = f\ (eval\ i\ q) = f\ (Indet\ m) = Indet\ (f\ m)$. This implies that $eval\ (f\ i)\ p \neq eval\ (f\ i)\ q$. From this and the semantics of $\leftrightarrow$ follows $eval\ (f\ i)\ (p \leftrightarrow q) = \circ$. Likewise, from $eval\ i\ p \neq eval\ i\ q$ and the semantics of $\leftrightarrow$ follows $eval\ i\ (p \leftrightarrow q) = \circ$. These two facts allow us to establish $eval\ (f\ i)\ (p \leftrightarrow q) = \circ = f\ \circ = f\ (eval\ i\ (p \leftrightarrow q))$. And this part of the proof is done. This was just one out of the 17 cases mentioned above. For the rest I refer to the formalization. ◀

Writing out all 17 cases mentioned above would be tedious and checking all of them by hand requires discipline. Therefore, there is always the danger of overlooking a needed argument, because one case looked similar to another but really was not. Formalization enforces this discipline.

Now it is time to prove that if there are at least as many indeterminate truth values in U as the number of propositional symbols in p, then the validity of p in $\mathbb{V}_U$ implies the validity of p in $\mathbb{V}$. The lemma is expressed using Isabelle/HOL's *card* function, which for finite sets returns their cardinality and for infinite sets returns 0.

theorem *valid_in_valid*:
assumes "*card U* ≥ *card* (*props p*)"
assumes "*valid_in U p*"
shows "*valid p*"

Proof. p is proved valid by fixing an arbitrary interpretation i: First, obtain an injection f of type $nat \Rightarrow nat$ such that f maps any value in i ' (*props p*) to a value in *domain U*. This is possible because $|domain\ U| \geq |props\ p|$.

Now define the following interpretation:

$$i'\ s = \begin{cases} (f\ i)\ s & \text{if } s \in props\ p \\ \bullet & \text{otherwise} \end{cases}$$

From the properties of f and definition of i' it follows that $range\ i' \subseteq domain\ U$ and then by the validity of p in U it follows that $eval\ i'\ p = \bullet$. Furthermore, i' and $f\ i$ coincide on all

symbols in p, and therefore, by the lemma *relevant_props*, it also follows that $eval\ (f\ i)\ p = \bullet$. Now from *eval_change* follows that $f\ (eval\ i\ p) = \bullet$. By definition of the application of a $nat \Rightarrow nat$ to a truth-value it is the case that $eval\ i\ p = \bullet$. Thus any interpretation evaluates to $\bullet$ and therefore the formula is valid. ◀

theorem *valid_iff_valid_in*:
assumes "*card U* ≥ *card* (*props p*)"
shows "*valid p* ⟷ *valid_in U p*"

Proof. Follows from *valid_valid_in* and *valid_in_valid*. ◀

5 Sets of Equal Cardinality Define the Same Logic

Recall that while the indeterminate values are all different and can be told apart using ⇔, none of them play any special role compared to the others. Therefore one would expect $\mathbb{V}_U$ and $\mathbb{V}_W$ to be the same when U and W have the same cardinality. In the same way, consider what happens when $|U| < |W|$. In this case one can think of $\mathbb{V}_U$ as being $\mathbb{V}_W$ with some truth values, and thus interpretations, removed. Removing interpretations only makes it easier for a formula to be valid and thus any formula that is valid in $\mathbb{V}_W$ should also be valid in $\mathbb{V}_U$.

Isabelle/HOL defines *inj_on* such that *inj_on f A* expresses that f is an injection from A into the return type of f. In order to be able to talk about one set having smaller cardinality than another, it is useful to also define the notion of an injection from a set into another set.

definition *inj_from_to* :: "('*a* ⇒ '*b*) ⇒ '*a set* ⇒ '*b set* ⇒ *bool*" **where**
"*inj_from_to f X Y* ≡ *inj_on f X* ∧ *f* ' *X* ⊆ *Y*"

The lemma *eval_change* is generalized from the type *nat* to sets of *nat*s.

lemma *eval_change_inj_on*:
assumes "*inj_on f U*"
assumes "*range i* ⊆ *domain U*"
shows "*eval* (*f i*) *p* = *f* (*eval i p*)"

Proof. The proof is analogous to that of *eval_change*. ◀

This is enough to prove the following lemma:

lemma *inj_from_to_valid_in*:
assumes "*inj_from_to f W U*"
assumes "*valid_in U p*"
shows "*valid_in W p*"

Proof. The plan is to fix an arbitrary interpretation in $\mathbb{V}_W$ and prove that it makes p true. First, realize that $range\ (f\ i) \subseteq domain\ U$; this follows from the fact that for any x it is the case that $(f\ i)\ x = f\ (i\ x)$ and here the application of i will give an element in $domain\ W$ and then the application of f will give an element in $domain\ U$. Therefore $eval\ (f\ i)\ p = \bullet$ by the validity of p in $\mathbb{V}_U$. Then use *eval_change_inj_on* to get that $f\ (eval\ i\ p) = \bullet$ and then from the definition of the application of f to a truth value that $eval\ i\ p = \bullet$. ◀

It is now time to prove that if U and W have equal cardinality, they define the same logic.

lemma *bij_betw_valid_in*:
assumes "*bij_betw f U W*"
shows "*valid_in U p* ⟷ *valid_in W p*"

Proof. f is an injection from U into W. f^- is an injection from W into U. The lemma therefore follows from *inj_from_to_valid_in*. ◀

6 The Difference Between $\mathbb{V}$ and $\mathbb{V}_U$ for a Finite U

Section 4 showed that the question of the validity of p in $\mathbb{V}$ can be reduced to the question of its validity in $\mathbb{V}_{\{0..<|prop\ p|\}}$, where $\{n..<m\} = \{k \mid n \leq k < m\}$ for any n and m. This section shows that this does not mean that $\mathbb{V}$ collapses to a finite valued $\mathbb{V}_U$. The approach is to demonstrate a formula that is true in $\mathbb{V}_{0..n}$ but false in $\mathbb{V}$. The formula is called the pigeonhole formula. For $n = 3$ the pigeonhole formula π_3 is

$$\pi_3 = \nabla \mathsf{x}_0 \mathbin{⩓} \nabla \mathsf{x}_1 \mathbin{⩓} \nabla \mathsf{x}_2 \Rightarrow (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_1) \mathbin{⩔} (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_2) \mathbin{⩔} (\mathsf{x}_0 \Leftrightarrow \mathsf{x}_1).$$

I.e. it states that, assuming that x_0, x_1 and x_2 refer to indeterminate values, two of them will be the same. This is of course not true in an interpretation where they map to three different values, but if one only considers two indeterminate values there are no such interpretations. Therefore the formula is not valid in general but it is valid in $\mathbb{V}_{\{\mathsf{I},\ \mathsf{II}\}}$. Propositions x_0 and x_1 and x_2 can be thought of as pigeons and the values I and II as pigeonholes.

In order to define the formula for any n, first define the conjunction and disjunction of any list $[p_1, \ldots, p_n]$ of formulas:

$$[⩓][p_1, \ldots, p_n] = p_1 \mathbin{⩓} \cdots \mathbin{⩓} p_n$$

$$[⩔][p_1, \ldots, p_n] = p_1 \mathbin{⩔} \cdots \mathbin{⩔} p_n$$

Extend $\nabla$to a symbol that characterizes lists of indeterminate values:

$$[\nabla][p_1, \ldots, p_n] = [⩓][\nabla p_1, \ldots, \nabla p_n]$$

Given two sets S_1 and S_2, the concept of their cartesian product $S_1 \times S_2$ is well known. Their *off-diagonal product* is defined as

$$S_1 \times_{\text{off-diag}} S_2 = \{(s_1, s_2) \in S_1 \times S_2 \mid s_1 \neq s_2\}$$

Isabelle/HOL offers the function *List.product* of type $'a\ list \Rightarrow 'a\ list \Rightarrow ('a \times 'a)\ list$, which implements the cartesian product on lists representing sets. From this the *list off-diagonal product* is defined:

$$L_1 \times_{\text{off-diag}} L_2 = \mathit{filter}\ (\lambda(x, y).\ x \neq y)\ (\mathit{List.product}\ L_1 L_2)$$

The list off-diagonal product is used to introduce equivalence existence, which given a list of formulas expresses that two of the formulas in the list are equivalent.

$$[\exists\,{=}][p_1, \ldots, p_n] = [⩔]([=]((p_1, \ldots, p_n) \times_{\text{off-diag}} (p_1, \ldots, p_n)))$$

where

$$[=][(p_{11}, p_{12}), \ldots, (p_{n1}, p_{n2})] = p_{11} \Leftrightarrow p_{12}, \ldots, p_{n1} \Leftrightarrow p_{n2}$$

Let $\mathsf{x}_0, \mathsf{x}_1, \mathsf{x}_2, \ldots$ be a sequence of different variables. These will form the pigeonholes. Implication, ∇, equivalence existence and the pigeonholes are combined to form the pigeonhole formula:

$$\pi_n = [\nabla][\mathsf{x}_0, \cdots, \mathsf{x}_{n-1}] \Rightarrow [\exists\,{=}][\mathsf{x}_0, \cdots, \mathsf{x}_{n-1}]$$

6.1 π_n is not valid in $\mathbb{V}$

In order to prove that the pigeonhole formula is not valid, a counter-model for it is demonstrated. This counter-model is in $\mathbb{V}_{\{0..<n\}}$ and is thus also a counter-model for the validity of the pigeonhole formula in $\mathbb{V}_{\{0..<n\}}$. The counter-model for pigeonhole formula number n is

$$c_n(y) = \begin{cases} \mathit{Indet}\ i & \text{if } y = \mathsf{x}_i \text{ and } i < n \\ \bullet & \text{otherwise} \end{cases}$$

In order to prove that it indeed is a counter-model of the pigeonhole formula, a number of lemmas are introduced that characterize the semantics of the formula's components:

lemma *cla_false_Imp*:
assumes "*eval i a* = •"
assumes "*eval i b* = ∘"
shows "*eval i* (*a* ⇒ *b*) = ∘"

Proof. Follows directly from the involved definitions. ◀

lemma *eval_CON*:
"*eval i* ([⩓] *ps*) = *Det* (∀ *p* ∈ *set ps*. *eval i p* = •)"

Proof. Note that *set ps* denotes the set of members in the list *ps*. The lemma follows by induction on the list *ps* from the involved definitions. ◀

lemma *eval_DIS*:
"*eval i* ([⩔] *ps*) = *Det* (∃ *p* ∈ *set ps*. *eval i p* = •)"

Proof. Follows by induction on the list *ps* from the involved definitions. ◀

lemma *eval_ExiEql*:
"*eval i* ([∃ =] *ps*) = *Det* (∃ (*p*₁, *p*₂)∈(*set ps* ×$_{\mathit{off\text{-}diag}}$ *set ps*). *eval i p*₁ = *eval i p*₂)"

Proof. Follows from the definition of [∃ =], the definition of $\times_{\text{off-diag}}$ and *eval_DIS*. ◀

is_indet t is defined to be true iff *t* is indeterminate. Likewise *is_det t* is true iff *t* is determinate.

lemma *eval_Nab*: "*eval i* (**∇** *p*) = *Det* (*is_indet* (*eval i p*))"

Proof. Follows directly from the involved definitions. ◀

lemma *eval_NAB*:
"*eval i* ([**∇**] *ps*) = *Det* (∀ *p* ∈ *set ps*. *is_indet* (*eval i p*))"

Proof. Follows from the definition of [**∇**], *eval_CON* and *eval_Nab*. ◀

With this one can prove that the pigeonhole formula is false under the c_n counter-model.

lemma *interp_of_id_pigeonhole_fm_False*: "*eval* c_n π_n = ∘"

Proof. The lemma *cla_false_Imp* states that an implication can be proved false by proving its antecedent true and conclusion false. Start by proving the antecedent true: The antecedent is $[\nabla][\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}]$, and this means that all the variables in $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ should refer to indeterminate values, which indeed they do by the definition of c_n. The conclusion $[\exists =][\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}]$ is proved false using *eval_ExiEql*, which reduces the problem to proving that no pair of different symbols among $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ evaluate to the same. That follows from how c_n is defined. ◀

From this follows that the pigeonhole formula is not valid:

theorem *not_valid_pigeonhole_fm*: "¬ *valid* π_n"

Proof. Follows from *interp_of_id_pigeonhole_fm_False*. ◀

It follows that the pigeonhole formula is not valid in $U_{\{0..<n\}}$:

theorem *not_valid_in_n_pigeonhole_fm*: "¬ *valid_in* $\{0..<n\}$ π_n"

Proof. From c_n's definition follows that *range* $c_n \subseteq$ *domain* $\{0..<n\}$. It follows that π_n is not valid in $U_{\{0..<n\}}$ by *interp_of_id_pigeonhole_fm_False* and the definition of validity in $U_{\{0..<n\}}$ ◀

6.2 π_n is valid in $\mathbb{V}_{\{0..<m\}}$ for $m < n$

In order to prove that π_n is valid in $\mathbb{V}_{\{0..<m\}}$ for $m < n$, a new lemma on the semantics of an implication is needed:

lemma *cla_imp_I*:
assumes "*is_det* (*eval i a*)"
assumes "*is_det* (*eval i b*)"
assumes "*eval i a* = • ⟹ *eval i b* = •"
shows "*eval i* (*a* ⇒ *b*) = •"

Proof. Not surprisingly, it follows directly from the involved definitions. ◀

∇ and $[\exists =]$ returning determinate values is also needed.

lemma *is_det_NAB*: "*is_det* (*eval i* ([∇] *ps*))"

Proof. The lemma follows from *eval_NAB*. ◀

lemma *is_det_ExiEql*: "*is_det* (*eval i* ([$\exists =$] *ps*))"

Proof. The lemma follows from *eval_ExiEql*. ◀

Moreover the pigeonhole principle is needed. This theorem is part of Isabelle/HOL in the following formulation:

lemma *pigeonhole*: "*card A* > *card* (*f* ‘ *A*) ⟹ ¬ *inj_on f A*"

It states that if the image of f on A is of smaller cardinality than A, then f cannot be an injection. From this follows a more specific formulation of the principle, which will be applied:

lemma *pigeon_hole_nat_set*:
assumes "*f* ‘ $\{0..<n\}$ ⊆ $\{0..<m\}$"
assumes "*m* < (*n* :: *nat*)"
shows "∃ j_1∈$\{0..<n\}$. ∃ j_2∈$\{0..<n\}$. $j_1 \neq j_2 \wedge f\, j_1 = f\, j_2$"

Proof. From the assumptions follows that *card* $\{0..<n\}$ > *card* $\{0..<m\}$ ≥ *card* (*f* ‘ $\{0..<n\}$). Therefore *pigeonhole* is applicable and the conclusion follows immediately. ◀

The pigeonhole formula will evaluate to true in any interpretation with truth values in $\mathbb{V}_{\{0..m\}}$ where $m < n - 1$:

lemma *eval_true_in_lt_n_pigeonhole_fm*:
assumes "$m < n$"
assumes "*range* $i \subseteq$ *domain* $\{0..<m\}$"
shows "*eval* $i\ \pi_n = \bullet$"

Proof. Apply *cla_imp_I* to break down the conclusion. The two first assumptions of *cla_imp_I* follow from *is_det_NAB* and *is_det_ExiEql*, and then what remains is to prove that the antecedent of π_n implies the conclusion of π_n. Therefore, assume that the antecedent, $[\nabla][\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}]$, evaluates to true. From this and *eval_NAB* follows that $\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}$ all evaluate to indeterminate values. This, together with the fact that the range of i is *domain* $\{0..<m\}$, means that i must map any x_l where $l \in \{0..<n\}$ to *Indet* k for some $k \in \{0..<m\}$. Therefore, by *pigeonhole_nat_set* there are $j_1 < n$ and $j_2 < n$ such that x_{j_1} and x_{j_2} are different but i evaluates them to the same value. This is by *eval_ExiEql* exactly what is required for the conclusion $[\exists =][\mathsf{x}_0, \ldots, \mathsf{x}_{n-1}]$ to evaluate to true. ◀

Therefore the pigeonhole formula must be valid in $\mathbb{V}_{\{0..<m\}}$.

theorem *valid_in_lt_n_pigeonhole_fm*:
assumes "$m<n$"
shows "*valid_in* $\{0..<m\}$ (*pigeonhole_fm* n)"

Proof. Follows immediately from *eval_true_in_lt_n_pigeonhole_fm*. ◀

There are many other finite sets than $\{0..<m\}$. It is therefore desirable to extend the theorem to claim that π_n is valid in any V_U where $|U| < n$. This can be done using the result from Section 5:

theorem *valid_in_pigeonhole_fm_n_gt_card*:
assumes "*finite* U"
assumes "*card* $U < n$"
shows "*valid_in* U (*pigeonhole_fm* n)"

Proof. Follows from *valid_in_lt_n_pigeonhole_fm* and *bij_betw_valid_in* ◀

6.3 $\mathbb{V}$ is different from $\mathbb{V}_U$ where U is finite

The previous subsection demonstrated that π_n is valid in e.g. $\mathbb{V}_U$ where $|U| = n$ but not in $\mathbb{V}$. Therefore the logics are different:

theorem *extend*: "*valid* $\neq$ *valid_in* U" **if** "*finite* U"

Proof. Follows from *valid_in_pigeonhole_fm_n_gt_card* and *not_valid_pigeonhole_fm*. ◀

This can be seen as a justification of the infinitely many values in the logic – they cannot once and for all be replaced by a finite subset. The reduction in Section 4 only worked because there the size of U depended on the considered formula.

7 Discussion and Related Work

My previous paper with Villadsen [24] contains a thorough discussion of related work giving an overview of various many-valued logics that have been formalized in Isabelle/HOL. I will refrain from repeating the section here and mention again only the most pertinent works namely by Marcos [12] and Steen and Benzmüller [18]. Marcos developed an ML program

that can generate proof tactics; these tactics implement tableaux that can prove theorems in various finitely many-valued logics. Steen and Benzmüller defined a shallow embedding of the many-valued SIXTEEN logic into classical HOL. That the embedding is shallow means that the authors give formulas in SIXTEEN meaning by translating them to logical expressions of classical HOL. The authors can then use a theorem prover for HOL to prove these formulas. Benzmüller and Woltzenlogel Paleo [5] used the same approach to embed several higher-order modal logics and also showed the approach applied to a sketch of a paraconsistent logic. Several other logics have been embedded in HOL in this way, including conditional logics by Benzmüller, Gabbay, Genovese and Rispoli [2], quantified multimodal logics by Benzmüller and Paulson [3], first-order nominal logic by Steen and Wisniewski [26] and free logic by Benzmüller and Scott [4]. In contrast, the formalization in this paper is a deep – rather than shallow – embedding of $\mathbb{V}$ i.e. formulas in the logic are expressed as values in HOL and a semantics is formalized that gives meaning to these formulas. This formalization thus defines datatypes for formulas and a semantics rather than a tableau or a translation.

Theorems stating that a logic cannot be characterized by finite-valued matrices are quite common in the literature on non-classical logics. For instance, Gödel [8] proved that intuitionistic logic cannot be characterized by finite-valued matrices and Dugundji [7] proved that neither can any of the modal logics S1-S5. Carnielli, Coniglio and Marcos [6] characterize the logics of formal inconsistency which are paraconsistent logics that have a so-called consistency operator, such as the $\mathbf{\Delta}$ operator of $\mathbb{V}$. The authors also prove that a number of these logics cannot be characterized by finite-valued matrices.

A noteworthy characteristic of the present formalization is that all proofs were built from the ground up in the proof assistant – they were not based on any preexisting proofs. Proof assistants make it very clear when a proof is finished, and one does not have to reread it over and over to see if everything adds up. Furthermore, in the development I tried out different definitions of the implication used in the pigeonhole formula and the proof assistant was very helpful in checking that the changes did not break any proofs. Proof assistants of course ensure correctness of proofs. Many times I stated lemmas and proved them directly in the proof assistant. Other times the insurance of correctness was a hindrance in that on the way to a correct proof it was helpful to state lemmas that were "mostly correct" and whose expressions "mostly type checked", i.e. I abstracted away from some of the details. This was often better done on a piece of paper than in the proof assistant. However, after this process was done, it was definitely worth returning to the proof assistant to see if the "mostly correct" proof held up to the challenge of being formalized and thus turned into a correct proof.

The propositional fragment of a paraconsistent infinite-valued higher-order logic has now been formalized. The formalization only considers the case where the logic has a countably infinite set of indeterminate truth values. It could also be interesting to prove and formalize theorems about what happens in case an uncountably infinite type of indeterminate truth values is allowed. This could be done by replacing *nat* in the definition of *tv* with some uncountably infinite type T. Another way would be to replace *nat* with a type variable that could then be instantiated with *nat* or T. With this in place, I conjecture it would be possible to prove that the formulas that are valid with respect to *nat* are the same as those that are valid with respect to an uncountably infinite type T. My argument in the one direction is that if the formula is valid in T then it must also be valid in *nat* since there is an injection from *nat* to T, and thus it should be possible to make a generalization of *inj_from_to_valid_in* that covers the case of uncountable infinity. In the other direction I would argue that since the cardinality of T is larger than any *props p* one should be able to reuse the proof of *valid_in_valid* to prove that if p is valid with respect to T then it is also valid with respect to *nat*.

Another obvious next step would be to formalize the whole paraconsistent higher-order logic. The basis of such an endeavor could be the formalizations of HOL Light in HOL Light and HOL4 by respectively Harrison [9] and Kumar et al. [11]. The challenge is to give a semantics to the language. In the formalization in HOL4 this is done by abstractly specifying set theory in HOL. The same specification could be used for giving a semantics to the paraconsistent higher-order logic.

8 Conclusion

This paper formalizes Villadsen's paraconsistent infinite-valued logic $\mathbb{V}$ and the $|U|$-valued logics $\mathbb{V}_U$ as well as proves and formalizes several meta-theorems of the logic. One meta-theorem shows that, for any formula, the question of its validity in $\mathbb{V}$ can be reduced to the question of its validity in $\mathbb{V}_U$ for a large enough finite U. The other meta-theorems, to my knowledge not previously proved or formalized, characterize how the number of truth-values affects truths of the logic. One of them shows that when $|U| = |W|$ then $\mathbb{V}_U$ has the same truths as $\mathbb{V}_W$. Another shows that for any finite U it is the case that $\mathbb{V}$ and $\mathbb{V}_U$ are different logics. The theory was developed in parallel with its formalization. This illustrates that proof assistants can be used as tools, not only for formalizing established results, but also for developing new results – in this case the meta-theory of a logic.

References

1 Seiki Akama, editor. *Towards Paraconsistent Engineering*, volume 110 of *Intelligent Systems Reference Library*. Springer, 2016.

2 Christoph Benzmüller, Dov Gabbay, Valerio Genovese, and Daniele Rispoli. Embedding and automating conditional logics in classical higher-order logic. *Annals of Mathematics and Artificial Intelligence*, 66(1):257–271, 2012.

3 Christoph Benzmüller and Lawrence C. Paulson. Quantified Multimodal Logics in Simple Type Theory. *Logica Universalis*, 7(1):7–20, 2013.

4 Christoph Benzmüller and Dana S. Scott. Automating Free Logic in HOL, with an Experimental Application in Category Theory. *Journal of Automated Reasoning*, January 2019.

5 Christoph Benzmüller and Bruno Woltzenlogel Paleo. Higher-Order Modal Logics: Automation and Applications. In Wolfgang Faber and Adrian Paschke, editors, *Reasoning Web (RW)*, volume 9203 of *LNCS*, pages 32–74. Springer, 2015.

6 Walter Carnielli, Marcelo E. Coniglio, and João Marcos. Logics of Formal Inconsistency. In D.M. Gabbay and F. Guenthner, editors, *Handbook of Philosophical Logic*, pages 1–93. Springer, 2007.

7 James Dugundji. Note on a Property of Matrices for Lewis and Langford's Calculi of Propositions. *J. Symbolic Logic*, 5(4):150–151, December 1940.

8 Kurt Gödel. Zum intuitionistischen Aussagenkalkül. *Akademie der Wissenschaften in Wien*, 69:65–66, 1932.

9 John Harrison. Towards self-verification of HOL Light. In Ulrich Furbach and Natarajan Shankar, editors, *International Joint Conference on Automated Reasoning (IJCAR)*, volume 4130 of *LNCS*, pages 177–191. Springer, 2006.

10 Andreas Schmidt Jensen and Jørgen Villadsen. Paraconsistent Computational Logic. In P. Blackburn, K. F. Jørgensen, N. Jones, and E. Palmgren, editors, *8th Scandinavian Logic Symposium: Abstracts*, pages 59–61. Roskilde University, 2012.

11 Ramana Kumar, Rob Arthan, Magnus O. Myreen, and Scott Owens. Self-Formalisation of Higher-Order Logic: Semantics, Soundness, and a Verified Implementation. *Journal of Automated Reasoning*, 56(3):221–259, 2016.

12 João Marcos. Automatic Generation of Proof Tactics for Finite-Valued Logics. In Ian Mackie and Anamaria Martins Moreira, editors, *Tenth International Workshop on Rule-Based Programming, Proceedings*, volume 21 of *Electronic Proceedings in Theoretical Computer Science*, pages 91–98. Open Publishing Association, 2010.

13 Graham Priest, Koji Tanaka, and Zach Weber. Paraconsistent Logic. In Edward N. Zalta, editor, *Stanford Encyclopedia of Philosophy*. Metaphysics Research Lab, Stanford University, summer 2018 edition, 2018. URL: `https://plato.stanford.edu/archives/sum2018/entries/logic-paraconsistent/`.

14 Anders Schlichtkrull. Formalization of a Paraconsistent Infinite-Valued Logic (short abstract). *Automated Reasoning in Quantified Non-Classical Logics – 3rd International Workshop – Program*, 2018. URL: `https://easychair.org/smart-program/FLoC2018/ARQNL-program.html`.

15 Anders Schlichtkrull. *Formalization of Logic in the Isabelle Proof Assistant*. PhD thesis, Technical University of Denmark, 2018. URL: `http://matryoshka.gforge.inria.fr/pubs/schlichtkrull_phd_thesis.pdf`.

16 Anders Schlichtkrull and Jørgen Villadsen. IsaFoL: Paraconsistency. `https://bitbucket.org/isafol/isafol/src/master/Paraconsistency/`.

17 Anders Schlichtkrull and Jørgen Villadsen. Paraconsistency. *Archive of Formal Proofs*, December 2016. , Formal proof development. URL: `http://isa-afp.org/entries/Paraconsistency.html`.

18 Alexander Steen and Christoph Benzmüller. Sweet SIXTEEN: Automation via Embedding into Classical Higher-Order Logic. *Logic and Logical Philosophy*, 25(4):535–554, 2016.

19 Jørgen Villadsen. Combinators for Paraconsistent Attitudes. In P. de Groote, G. Morrill, and C. Retoré, editors, *Logical Aspects of Computational Linguistics (LACL)*, volume 2099 of *LNCS*, pages 261–278. Springer, 2001.

20 Jørgen Villadsen. A Paraconsistent Higher Order Logic. In B. Buchberger and J. A. Campbell, editors, *Artificial Intelligence and Symbolic Computation (AISC)*, volume 3249 of *LNCS*, pages 38–51. Springer, 2004.

21 Jørgen Villadsen. Paraconsistent Assertions. In G. Lindemann, J. Denzinger, I. J. Timm, and R. Unland, editors, *Multi-Agent System Technologies (MATES)*, volume 3187 of *LNCS*, pages 99–113, 2004.

22 Jørgen Villadsen. Supra-logic: Using Transfinite Type Theory with Type Variables for Paraconsistency. *Logical Approaches to Paraconsistency, Journal of Applied Non-Classical Logics*, 15(1):45–58, 2005.

23 Jørgen Villadsen and Anders Schlichtkrull. Formalization of Many-Valued Logics. In H. Christiansen, M.D. Jiménez-López, R. Loukanova, and L.S. Moss, editors, *Partiality and Underspecification in Information, Languages, and Knowledge*, chapter 7. Cambridge Scholars Publishing, 2017.

24 Jørgen Villadsen and Anders Schlichtkrull. Formalizing a Paraconsistent Logic in the Isabelle Proof Assistant. In Abdelkader Hameurlain, Josef Küng, Roland Wagner, and Hendrik Decker, editors, *Transactions on Large-Scale Data- and Knowledge-Centered Systems (TLDKS)*, volume 10620 of *LNCS*, pages 92–122. Springer, 2017.

25 Zach Weber. Paraconsistent Logic, 2019. URL: `https://www.iep.utm.edu/para-log/`.

26 Max Wisniewski and Alexander Steen. Embedding of Quantified Higher-Order Nominal Modal Logic into Classical Higher-Order Logic. In Christoph Benzmüller and Jens Otten, editors, *ARQNL 2014. Automated Reasoning in Quantified Non-Classical Logics*, volume 33 of *EPiC Series in Computing*, pages 59–64. EasyChair, 2015. `doi:10.29007/dzc2`.

Normalization by Evaluation for Typed Weak λ-Reduction

Filippo Sestini
Functional Programming Laboratory, University of Nottingham, United Kingdom
http://www.cs.nott.ac.uk/~psxfs5
filippo.sestini@nottingham.ac.uk

Abstract

Weak reduction relations in the λ-calculus are characterized by the rejection of the so-called ξ-rule, which allows arbitrary reductions under abstractions. A notable instance of weak reduction can be found in the literature under the name *restricted reduction* or *weak λ-reduction*.

In this work, we attack the problem of algorithmically computing normal forms for λ^{wk}, the λ-calculus with weak λ-reduction. We do so by first contrasting it with other weak systems, arguing that their notion of reduction is not strong enough to compute λ^{wk}-normal forms. We observe that some aspects of weak λ-reduction prevent us from normalizing λ^{wk} directly, thus devise a new, better-behaved weak calculus λ^{ex}, and reduce the normalization problem for λ^{w} to that of λ^{ex}. We finally define type systems for both calculi and apply Normalization by Evaluation to λ^{ex}, obtaining a normalization proof for λ^{wk} as a corollary. We formalize all our results in Agda, a proof-assistant based on intensional Martin-Löf Type Theory.

2012 ACM Subject Classification Theory of computation → Lambda calculus

Keywords and phrases normalization, lambda-calculus, reduction, term-rewriting, Agda

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.6

Supplement Material https://github.com/fsestini/nbe-weak-stlc

Acknowledgements We thank Maria Emilia Maietti, Claudio Sacerdoti Coen, Thorsten Altenkirch, Andreas Abel, and Delia Kesner for helpful discussions and feedback on this work.

1 Introduction

The weak λ-calculus can be described informally as the $\lambda\beta$-calculus without the ξ rule – the congruence rule for λ-abstractions – shown below for the simply-typed case. Without any other modifications, this system is not *confluent* (or Church-Rosser). The property can be recovered with the addition of a substitution rule, labeled (σ) below, which gives rise to a confluent system.

$$\frac{\Gamma, x : A \vdash t \longrightarrow s : B}{\Gamma \vdash \lambda x.t \longrightarrow \lambda x.s : A \to B}\ (\xi) \qquad \frac{\Gamma, x : A \vdash t : B \qquad \Gamma \vdash a \longrightarrow b : A}{\Gamma \vdash t[a/x] \longrightarrow t[b/x] : B}\ (\sigma)$$

This particular notion of weak reduction was originally formulated by Howard [25], although presented differently, and was later studied by Çağman and Hindley [13] under the name of *weak λ-reduction*. On the one hand, the addition of (σ) restores confluence, but on the other hand it complicates the design of an algorithmic procedure to mechanically compute terms to normal form: despite the absence of the ξ-rule, conversion under abstractions is still provable via the σ-rule for a restricted class of redexes, the so called *weak redexes*. That is, contrary to the non-confluent weak λ-calculus, contractions *can* occur under λ-abstractions. This makes weak λ-reduction *crucially* different from, and more complicated than other weak calculi that implement weak-*head* reduction instead, for which a normalization algorithm never needs to reduce under binders.

These issues must be faced when attempting to prove the normalization theorem for a typed version of the weak λ-calculus, a goal that motivated the work presented here. Indeed, suppose we want to show that every well-typed term t reduces to its normal form, given by a normalization function $[\![t]\!]_\rho$ defined by structural recursion, and in particular such that $[\![\lambda x.t]\!]_\rho = \lambda x.[\![t]\!]_{\rho[x/x]}$. In the case of a term $\Gamma \vdash \lambda x.t : A \Rightarrow B$, we can obtain $\Gamma, x : A \vdash t \longrightarrow [\![t]\!]_{\rho[x/x]} : B$ by induction hypothesis, but then it is not clear how to proceed, since we cannot in general conclude $\Gamma \vdash \lambda x.t \longrightarrow \lambda x.[\![t]\!]_{\rho[x/x]} : A \Rightarrow B$ without ξ.

Another unpleasant aspect of weak λ-reduction is its *relative* notion of redex. Consider the term $t \equiv \lambda z.(\lambda x.y)\ z$. Its subterm $(\lambda x.y)\ z$ is a valid redex under standard β reduction, leading to the reduction $\lambda z.(\lambda x.y)\ z \longrightarrow \lambda z.y$, but is not a valid redex *of* t in the weak λ-calculus, since there is no way to express this reduction in terms of (β) and (σ). The same term $(\lambda x.y)\ z$, instead, does reduce as a subterm of $\lambda w.(\lambda x.y)\ z$, leading to $\lambda w.y$. This relative notion of redex complicates the definition of a normalization algorithm, since it is not clear from the syntactic structure alone whether or not a redex can be justified by the substitution rule and should be contracted.

1.1 Contributions

In this paper we study weak λ-reduction, and propose a way to algorithmically reduce terms to their normal form. We also give a constructive proof of normalization for a simply-typed λ-calculus with weak λ-reduction. More precisely, we make the following contributions:

- We compare the λ-calculus with weak λ-reduction, that we call λ^{wk}, with other *weak* calculi rejecting the ξ-rule. We precisely characterize λ^{wk} normal forms, compare them with the other systems, and show that the problem of normalizing weak λ-reduction does not seem to be reducible to normalization of these calculi, identifying what we think is an unexplored area in the literature;
- We define a new calculus of weak reductions, λ^{ex}, that is inspired by our characterization of λ^{wk} redexes and in particular makes them syntactically explicit (hence the name). This system recovers a version of the ξ-rule, and admits a standard normalization algorithm. We then show λ^{wk} and λ^{ex} equivalent, thus reducing the problem of normalization for the former to normalization for the latter. This provides the first published specification of a normalization algorithm for λ^{wk} in full detail;
- We define type systems for λ^{wk} and λ^{ex}, and prove the latter normalizing via the semantic method of Normalization by Evaluation [11]. Mirroring the untyped case, the two calculi are shown equivalent. Transporting the normalization proof for λ^{ex} along this equivalence yields, as far as we are aware, the first proof of normalization for the simply-typed λ-calculus with weak λ-reduction;
- We include a full formalization of this work in intensional Type Theory, using the Agda proof-assistant [12] [1].

1.2 Meta-theory and notation

We will use a rather informal and foundation-agnostic notation, that can be understood both in Type Theory and in constructive set theory. We do however distinguish between definitional equalities ($\equiv$), which are always decidable, and propositional equalities ($=$), for which decidability needs to be proved.

[1] The formalization can be found at `https://github.com/fsestini/nbe-weak-stlc`

▶ Remark 1. Readers interested in a proof-checked formalization of this work in Type Theory are invited to consult the Agda code, keeping in mind the following differences:

- The partial functions represented here with the usual function notation cannot be defined in Agda as standard type-theoretic functions, since these must be total in order to preserve logical consistency. In the formalization, partial functions are encoded by their graph, i.e. as inductively-defined functional (left-total, right-unique) relations;
- In the Agda code, we use a nameless [20] syntax to represent λ-terms.

2 Weak λ-reduction

The weak λ-calculus is generally defined as a flavor of the λ-calculus whose reduction relation does not include the weak extensionality principle represented by the ξ-rule [9], also referred to as the congruence rule for λ-abstractions. One of the simplest forms of weak reduction is obtained by stripping β-reduction of the ξ-rule.

▶ **Definition 2** (Weak reduction). *Weak reduction is inductively defined as follows:*

$$\frac{}{(\lambda x.t)s \longrightarrow t[s/x]}\ (\beta) \qquad \frac{t \longrightarrow r}{t\ s \longrightarrow r\ s}\ (\nu) \qquad \frac{s \longrightarrow r}{t\ s \longrightarrow t\ r}\ (\mu)$$

Weak reduction, also known as *weak-head reduction*, is of interest in the study of programming languages, as it captures the fact that evaluation of programs does not generally proceed under binders [23]. Weak reduction evaluates terms to *weak-head* normal form (*whnf*):

$$\mathsf{Whnf} \ni d ::= \lambda x.t \mid x d_1 d_2 ... d_n$$

Weak reduction never reduces under binders, and as a consequence the body t of a weak-head normal abstraction may be an arbitrary term, not necessarily a whnf. Unfortunately, this relation is not confluent, hence only specific weak reduction strategies have been studied in the literature [36, 4]. A solution to this problem consists of extending weak reduction with a primitive substitution rule [31], the σ-rule (Section 1).

One of the first uses of this confluent variant of weak reduction dates back to Howard [25], who calls it *restricted reduction*. Here we will refer to it as *weak λ-reduction* after Çağman and Hindley [13].

▶ **Definition 3** (Weak λ-reduction). *The reduction relation $\longrightarrow_w$ is defined as the relation in Definition 2 plus the σ-rule.*

▶ **Theorem 4.** *$\longrightarrow_w^*$, the reflexive-transitive closure of $\longrightarrow_w$, is confluent.*

Proof. See [31], Theorem 1. ◀

In [13], the authors cite an alternative formulation of weak λ-reduction based on the notion of *weak redex*, due to Howard:

▶ **Definition 5.** *Let the redex r be a subterm of a term t. Then, r is a* weak redex *if and only if it does not contain free variables that are bound in t. A one-step weak λ-contraction of t is one that contracts a weak redex inside t.*

For example, the term $\lambda x.(\lambda y.y)z$ contains the weak redex $(\lambda y.y)z$. Conversely, the term $\lambda x.(\lambda y.x)z$ has no weak redexes, and is therefore in normal form. These two definitions give rise to equivalent relations [13].

We call λ^{wk} the λ-calculus with weak λ-reduction, as given in Definition 3, and refer to its normal forms as *weak normal forms*. These are not characterized as easily as whnfs, and the reason is that being a weak normal form is a relative property, since being a weak redex is, and normal terms are just terms with no weak redexes. We observe that the essence of weak redexes can be reduced to the distinction between two *roles* for variables. Given a term t, we say that a variable has *local* role if it is bound somewhere within t, and *global* otherwise. Then, a weak redex is just a redex that is closed w.r.t. local variables (cfr. Definition 5.) More precisely, if $t \equiv C[r]$ for an *enclosing context* $C[\_]$ with a hole and a redex r, then r is a weak redex iff it is closed w.r.t. the local variables bound by abstractions within $C[\_]$. This suggests a notion of normal form that is indexed by the set V of local variables of the enclosing context, whatever this is. For convenience, we define normal terms Nf^V mutually with *neutral terms* Ne^V, both under a set of variables V:

$$\mathsf{Nf}^V \ni d^V ::= e^V \mid \lambda x.d^{V\cup\{x\}}$$

$$\mathsf{Ne}^V \ni e^V ::= x \mid e^V\, d^V \mid (\lambda x.d_1^{V\cup\{x\}})\, d_2^V \text{ s.t. } FV((\lambda x.d_1)d_2) \cap V \neq \emptyset$$

Neutral terms usually correspond to variables and elimination forms whose reduction is "blocked" by the presence of a neutral term in recursive position. In the standard $\lambda\beta$-calculus, neutral terms are only variables and "stuck" applications. In our setting, the definition of neutral term needs to be extended, to account for redexes of the form $(\lambda x.t)s$ that are not *weak*, and therefore "stuck" as well. Given a set V of local variables, this is the case whenever some variables in V are free in the redex, namely $FV((\lambda x.t)s) \cap V \neq \emptyset$.

We define the set of weak normal forms $\mathsf{Nf} :\equiv \mathsf{Nf}^\emptyset$. As an example, we can see that $\lambda xy.(\lambda z.x)w \in \mathsf{Nf}$, since $(\lambda z.x)w \in \mathsf{Ne}^{\{x,y\}}$, but $\lambda x.(\lambda y.y)z \notin \mathsf{Nf}$, since $FV((\lambda y.y)z) \equiv \{z\} \cap \{x\} = \emptyset$.

2.1 Algorithmic weak λ-reduction

Although typed β-reduction is normalizing [24], and we know that weak λ-reduction constitutes a (strict) subset of full β-reduction, these facts alone do not provide us with an algorithm to actually compute normal forms, or even ensure that such algorithm exists.

Here, we are interested in computational solutions to the problem of algorithmic reduction for (typed) calculi based on weak λ-reduction. A way to achieve this goal is to seek a *constructive* proof of the normalization theorem. When formalized in Intuitionistic Type Theory, this proof comes by its nature with a normalization algorithm for untyped terms baked-in. This is quite convenient, but it also means that we cannot hope to proceed any further without some understanding of how to go by implementing such algorithm. The presence of the substitution rule (σ) interacts with this goal in unpleasant ways:

- Despite the absence of the ξ-rule, *some* contractions can still happen under binders via (σ). A normalization algorithm will thus have to proceed, in some way, by recursion on the structure of terms and thus under λs; but this is at odds with weak λ-reduction not being a congruence relation, which prevents us from reasoning about these recursive reductions in an adequate way.
- Weak λ-reduction has a relative notion of redex, defined w.r.t. some term t that contains it as a subterm (Definition 5). What makes a redex *weak* is therefore not evident from its syntactic structure alone, so a standard normalization function that just proceeds by structural recursion does not seem sufficient.

Perhaps because of these difficulties, or perhaps because of the exotic nature of weak λ-reduction, we could not find a complete specification of a normalization algorithm for this notion of reduction anywhere in the literature, nor a proof of normalization for a typed calculus based on it. A first approach towards filling this gap is to try to reduce the problem of normalization for λ^{wk} to normalization of other weak calculi for which a solution is well-known. We attempt to do so in the next sections.

2.2 Combinatory logic

There is a correspondence between λ^{wk} and combinatory logic (CL) [18], made precise in [13] by means of two operations $\_\lambda$ and $\_H$ respectively translating CL terms to λ-terms and vice versa. $\_\lambda$ is the obvious translation of combinators to λ-terms, whereas $\_H$ is essentially combinator (or *bracket*) abstraction ([24], Definition 2.18). We can then show

▶ **Theorem 6.** *For all combinators c, d and λ-terms t, s:*

- $c \triangleright_w d \implies c_\lambda \longrightarrow_w d_\lambda$;
- $t \longrightarrow_w s \implies t_H \triangleright_w s_H$.

Proof. See [13]. ◀

Here $\triangleright_w$ is combinator reduction. Normalization for combinatory logic is well understood, and in particular Normalization by Evaluation has been successfully applied both to the typed [16] and the untyped [21] cases. Since our goal is to algorithmically reduce λ-terms to weak normal form, we may hope to be able to reduce the problem to that of normalizing combinators, by exploiting the correspondence stated in Theorem 6.

Unfortunately, normal forms are not correctly related by the two translation operations. A counter-example is given by the λ-term $t \equiv \lambda x.(\lambda y.x)(II)$, which has a weak normal form $\lambda x.(\lambda y.x)I$ (where I is the identity). The term t translates to $t_H \equiv SK(K(II))$, and its normal form $SK(KI)$ translates back to the λ-term $\lambda w.((\lambda xy.x)w)((\lambda xy.x)Iw)$. But this term is neither normal (since $(\lambda xy.x)I$ is a weak redex) nor convertible to $\lambda x.(\lambda y.x)I$ in the absence of ξ.

This mismatch is also observed in [40], where a version of Martin-Löf Type Theory is compared to a combinator-based formulation. We are not aware of a way to relate CL- and λ-terms that makes it possible to rely on combinatory reduction to fully compute weak λ-reduction.

2.3 Weak explicit substitutions

Explicit substitutions are a way to formulate the syntax and reduction rules of the λ-calculus that turns substitutions into constructors of the syntax of terms, and integrates the substitution operation as part of the reduction relation, rather than an implicit meta-theoretic operation [1]. There have been several attempts at modeling weak reduction with explicit substitutions [16, 17, 31, 7]. In that setting, the *weak* character of weak reduction can be captured by stipulating that substitutions should not be propagated under binders. We describe some attempts at doing this, starting with the weak λ-calculus by Lévy and Maranget [31], whose substitution mechanism is a hybrid between implicit and explicit:

$$\mathsf{Term} \ni t, s ::= x \mid ts \mid (\lambda x.p)[\sigma] \qquad \mathsf{Prog} \ni p, q ::= x \mid pq \mid \lambda x.p$$
$$\mathsf{Subst} \ni \sigma ::= (x_1, t_1), (x_2, t_2), \ldots, (x_n, t_n)$$

Here, we use the metavariables p, q for *programs*, i.e. constant terms, and t, s for ordinary *terms*. Explicit substitutions σ are just lists of variable-term pairs. We write $\sigma[t/x]$ for the extended substitution that maps x to t and behaves like σ otherwise. Terms are formed out of variables, application, and *closures* $(\lambda x.p)[\sigma]$, i.e. pairs made of a functional program $\lambda x.p$ and a substitution σ assigning terms to its free variables. However, we do have implicit substitution $\langle \_ \rangle$ on programs:

$$x\,\langle\sigma\rangle :\equiv \sigma(x) \qquad (p\ q)\langle\sigma\rangle :\equiv p\langle\sigma\rangle\ q\langle\sigma\rangle \qquad (\lambda x.p)\langle\sigma\rangle :\equiv (\lambda x.p)[\sigma]$$

Here $\sigma(x)$ just looks up the term associated to the variable x in a substitution. The dynamics of weak explicit substitutions is defined as follows; closures are used to avoid pushing substitutions under λ-abstractions, since otherwise we would validate the ξ-rule.

▶ **Definition 7** (Weak explicit substitutions)**.**

$$\frac{}{(\lambda x.p)[\sigma]\ s \longrightarrow p\langle\sigma[s/x]\rangle}\ (\beta) \qquad \frac{\sigma \longrightarrow \sigma'}{(\lambda x.p)[\sigma] \longrightarrow (\lambda x.p)[\sigma']}\ (\xi\text{-}subst)$$

$$\frac{t \longrightarrow t'}{ts \longrightarrow t's}\ (\nu) \qquad \frac{s \longrightarrow s'}{ts \longrightarrow ts'}\ (\mu) \qquad \frac{t \longrightarrow s}{\sigma, (x_i, t), \sigma' \longrightarrow \sigma, (x_i, s), \sigma'}$$

A normal form is thus either a variable applied to normal forms, or a functional program together with a normal substitution:

$$\mathsf{Nf} \ni d ::= x\,d_1 \ldots d_n \mid (\lambda x.p)[\rho] \qquad \mathsf{Nfs} \ni \rho ::= (x_1, d_1), (x_2, d_2), \ldots, (x_n, d_n) \tag{1}$$

A related calculus was defined by Martin-Löf in one of the early formulations of his type theory [35]. In that system, terms of function type are not constructed by abstraction, but by introducing a fresh symbol for a combinator. The system includes a primitive substitution rule like the σ-rule in Definition 3, but since functions are just atomic symbols, substitution *de facto* never happens under binders. We do not reproduce Martin-Löf's system here, but instead consider an alternative formulation due to Coquand and Dybjer [16], which is completely equivalent for the purpose of our analysis. The system is based on explicit substitutions and is very similar to that of Definition 7, although they present it using a typed nameless syntax [20]:

$$\mathsf{Term} \ni t, s ::= x \mid ts \mid (\lambda x.t)[\sigma] \qquad \mathsf{Subst} \ni \sigma ::= (x_1, t_1), \ldots, (x_n, t_n)$$

The normal forms can be characterized in the same way as (1), with the only difference that now λ-abstractions in normal form $(\lambda x.t)[\rho]$ have ordinary terms as their body. A further generalization is obtained by allowing explicit substitutions on any term instead of just on functional closures. An example is the *weak $\lambda\sigma$-calculus* of [17], which we will call $\lambda^w\sigma$, and that is also considered in its typed nameless version in [16].

$$\mathsf{Term} \ni t, s ::= x \mid ts \mid \lambda x.t \mid t[\sigma] \qquad \mathsf{Subst} \ni \sigma ::= \langle\rangle \mid (x, t), \sigma \mid \sigma_1 \cdot \sigma_2$$

Here Subst includes an empty substitution $\langle\rangle$, and a composition constructor $\sigma_1 \cdot \sigma_2$, which is used to model terms under multiple substitutions: $t[\sigma_1][\sigma_2] \longrightarrow t[\sigma_1 \cdot \sigma_2]$. Reduction is essentially that of Definition 7, apart from the β-rule which is now $(\lambda x.t)[\sigma]s \longrightarrow t[(x, s), \sigma]$, and the addition of reduction rules for explicit substitutions. Normal forms are characterized exactly as in Martin-Löf's weak λ-calculus just described, namely

$$\mathsf{Nf} \ni d ::= x\,d_1 \ldots d_n \mid (\lambda x.t)[\rho] \qquad \mathsf{Nfs} \ni \rho ::= (x_1, d_1), (x_2, d_2), \ldots, (x_n, d_n)$$

These calculi of weak explicit substitutions are very similar, and in particular they all implement some form of *weak-head* reduction, where computation does not occur under binders. Weak-head normalization is relatively simple and well-understood, and both [35] and [16] include proofs of normalization for their respective systems. We now compare weak explicit substitutions to λ^{wk}, particularly to see whether these can be used to compute λ^{wk}-normal forms. We consider $\lambda^{w}\sigma$ as a representative of the calculi of weak explicit substitutions that we have presented, as it can simulate the others.

Note that terms of the weak λ-calculus can be embedded into $\lambda^{w}\sigma$, so a naive approach would be to just treat weak λ-terms as terms of $\lambda^{w}\sigma$, and normalize them under this reduction relation. $\lambda^{w}\sigma$ normal forms can be turned back into regular λ-terms by just fully applying explicit substitutions as they were implicit. That this translation fails to achieve our goal is already evident by observing the difference in the normal forms of the two calculi. In particular, every λ-abstraction is a normal form in $\lambda^{w}\sigma$, even when its body is not normal, whereas in the implicit weak calculus, abstractions are only normal if they do not contain weak redexes, that is, $\lambda x.d \in \mathsf{Nf} \iff d \in \mathsf{Nf}^{\{x\}}$.

This does not necessarily settle the question negatively, because there could be a way to translate λ^{wk}-terms to $\lambda^{w}\sigma$-terms in a way that makes this method work. In [31], the authors consider a translation via maximal free subterms. A subterm t' of a term t is free whenever $t \equiv C[t']$ for some context $C[\_]$ that does not bind any free variable in t'. A free subterm is maximal whenever it is not a subterm of another free subterm. We define a translation operation $\mathcal{T}_1$ from λ^{wk} to $\lambda^{w}\sigma$:

$$\mathcal{T}_1(x) = x \qquad \mathcal{T}_1(ts) = \mathcal{T}_1(t)\mathcal{T}_1(s)$$
$$\mathcal{T}_1(\lambda x.t) = (\lambda x.C[x_1, \dots, x_n])[(x_1, \mathcal{T}_1(t_1)), \dots, (x_n, \mathcal{T}_1(t_n))]$$

where $t_1, \dots, t_n$ are the maximal free subterms of t. We also define $\mathcal{T}_2$ as the converse translation that again just applies all explicit substitutions. We can now show that a reduction in λ^{wk} translates to a reduction in $\lambda^{w}\sigma$:

▶ **Proposition 8.** *If $t \longrightarrow_w \mathcal{T}_2(s)$, then $\mathcal{T}_1(t) \longrightarrow s$.*

Unfortunately, this result does not generalize to the reflexive-transitive closure of reduction, so in particular it fails to relate normal forms in the two calculi as we require. In fact, consider the following reduction of a term t in λ^{wk}:

$$t \equiv (\lambda x.\lambda y.xz)(\lambda w.w) \longrightarrow_w \lambda y.(\lambda w.w)z \longrightarrow_w \lambda y.z$$

On the other hand, we have

$$\mathcal{T}_1(t) \equiv (\lambda x.\lambda y.xk)[z/k](\lambda w.w) \longrightarrow (\lambda y.xk)[z/k, (\lambda w.w)/x] \equiv s$$

The term s is a normal form in $\lambda^{w}\sigma$, but translates to the reducible term $\mathcal{T}_2(s) \equiv \lambda y.(\lambda w.w)z$ in λ^{wk}. The problem is that Proposition 8 only holds for terms that are image of $\mathcal{T}_1$, i.e. those terms t such that their explicit substitutions only contain maximal free subterms in $\mathcal{T}_2(t)$. Unfortunately, β-contraction destroys this maximality property, since it can create new weak redexes in $\mathcal{T}_2(t)$ that do not exist in t. We conjecture that there exists a different pair of translation functions that makes this work, but leave the rigorous investigation of this aspect to future work.

3 Two-variable syntax

As anticipated at the end of Section 2, the notion of a weak redex in a term t is essentially about the distinction between two variable *roles*: the *local* variables that may appear bound

within t, and the *global* variables that may not. Therefore, the only information that is really needed by a normalization algorithm to reduce t to weak normal form is the role of all variables in t. We can exploit this fact by choosing a representation of λ-terms where this variable distinction is made syntactically explicit. Deciding whether a redex in t is weak then becomes a straightforward syntactic check. This is reminiscent of the *locally-nameless* representation of λ-terms [14], where a similar syntactic distinction is made between free and bound variables. We define two-variable λ-terms Term as follows

$$\mathsf{Term} \ni t, s ::= x^G \mid x^L \mid \lambda x.t \mid ts$$

We distinguish between global variables x^G and local variables x^L. Abstraction and application are the usual ones. We define the set of *all* free variables $FV(t)$ for a term t in the usual way ([9], Definition 2.1.7), and consider the obvious restrictions FV^L, FV^G to, respectively, local and global variables. We say that a term is *locally-closed* when it contains no free local variables, and define the set of locally-closed terms as $\mathsf{LC} \equiv \{t \in \mathsf{Term} \mid FV^L(t) = \emptyset\}$.

We consider two substitution operations $\_[\_/\_]$ and $\_\langle\_/\_\rangle$, dedicated respectively to local and global variables.

$$x^G\langle a/y\rangle :\equiv \begin{cases} a, & \text{if } x = y \\ x^G, & \text{otherwise} \end{cases} \qquad x^G[a/y] :\equiv x^G$$

$$x^L\langle a/y\rangle :\equiv x^L \qquad x^L[a/y] :\equiv \begin{cases} a, & \text{if } x = y \\ x^L, & \text{otherwise} \end{cases}$$

$$(\lambda x.t)\langle a/y\rangle :\equiv \lambda x.t\langle a/y\rangle \qquad (\lambda\, t)[a/y] :\equiv \lambda x.t[a/y]$$

$$(t\, s)\langle a/x\rangle :\equiv t\langle a/x\rangle\, s\langle a/x\rangle \qquad (t\, s)[a/y] :\equiv t[a/y]\, s[a/y]$$

We assume α-conversion is applied when needed to avoid variable capture, as well as the Barendregt convention ([9], 2.1.13). The two-variable syntax is instrumental in the definition of an auxiliary reduction relation, given in Definition 10 below, later shown equivalent to weak λ-reduction (Theorem 13). However, to facilitate the proof of this equivalence, we restate weak λ-reduction of Definition 3 in terms of the same two-variable syntax, and use this definition from now on. Since the local vs global variable distinction is irrelevant in this case, we restrict our definition to locally-closed terms. For this class of terms, the global (resp. local) variables end up corresponding to free (resp. bound) variables.

▶ **Definition 9** (Two-variable weak λ-reduction)**.** *The binary relation* $\_ \longrightarrow_w \_$ *between* locally-closed *two-variable terms is inductively defined as follows.*

$$\frac{}{(\lambda x.t)\, s \longrightarrow_w t[s/x]}\ (\beta) \qquad \frac{a \longrightarrow_w b}{t\langle a/x\rangle \longrightarrow_w t\langle b/x\rangle}\ (\sigma)$$

It can be seen that Definition 9 is just weak λ-reduction of Definition 3, modulo decorated variables and congruence rules, which are derivable from (σ). From now on, we will consider λ^{wk} to be the system with LC as terms and Definition 9 as reduction relation.

We now take advantage of the two-variable representation to define a normalization algorithm for the two-variable syntax, that will end up corresponding to normalization under weak λ-reduction. We specify it as a recursive traversal on arbitrary two-variable terms, not necessarily locally-closed. In particular, whenever we recursively reduce under a λ-abstraction, we do *not* replace the previously bound local variable with some free global one, but instead we leave it local. As a consequence, when the algorithm is applied to a locally-closed term $t \equiv C[s]$, recursive calls are able to identify which variables that appear free in a subterm s of t are bound by $C[\_]$, and thus which redexes are weak redexes, *without* having to keep track of what $C[\_]$ is.

A normalization function based on these ideas is shown below. We will use *parallel substitutions* $\rho \in \mathsf{Subst}$ to map free local variables to normal terms. These are defined as either an empty substitution $\langle\rangle$, or the extension $\rho[t/x]$ of ρ with the mapping $x \mapsto t$. We write $\rho(x)$ for the (partial) look-up function for the term associated to a variable, defined recursively in the obvious way, $[t/x]$ for the singleton parallel substitution, and $t[\rho]$ for the repeated application of all substitutions in ρ to a term t. Thus, if t is a locally-closed term, the expression $[\![t]\!]\langle\rangle$ gives the normal form of t, if it exists.

$$
\begin{aligned}
[\![x^G]\!]\,\rho &:\equiv x^G \\
[\![x^L]\!]\,\rho &:\equiv \rho(x) \\
[\![\lambda x.t]\!]\,\rho &:\equiv \lambda x.[\![t]\!]\,(\rho[x^L/x]) \\
[\![t\,s]\!]\,\rho &:\equiv [\![t]\!]\,\rho \bullet [\![s]\!]\,\rho
\end{aligned}
\qquad
t \bullet s :\equiv \begin{cases} [\![t']\!][s/x] & \text{if } t \equiv \lambda x.t' \text{ and } t\,s \in \mathsf{LC} \\ t\,s & \text{otherwise} \end{cases}
$$

Note that this normalization function takes untyped λ-terms as arguments, so it is necessarily *partial*, since not every untyped term admits a normal form under weak λ-reduction. As a consequence of this, in the Agda formalization untyped normalization is not defined as a type-theoretic function, but rather as an inductive functional relation (see remark in Section 1.2).

We can distill the function above into a reduction relation $\longrightarrow_{ex}$, that we call *explicit* because it makes use of the explicitly syntactic distinction between variable roles.

▶ **Definition 10** (Explicit two-variable weak λ-reduction). *The binary relation* $\_ \longrightarrow_{ex} \_$ *between* arbitrary *two-variable terms is defined as follows.*

$$
\frac{(\lambda x.t)\,s \in \mathsf{LC}}{(\lambda x.t)\,s \longrightarrow_{ex} t[s/x]}\ (\beta) \qquad \frac{t \longrightarrow_{ex} s}{\lambda x.t \longrightarrow_{ex} \lambda x.s}\ (\xi) \qquad \frac{t \longrightarrow_{ex} r}{t\ s \longrightarrow_{ex} r\ s}\ (\nu) \qquad \frac{s \longrightarrow_{ex} r}{t\ s \longrightarrow_{ex} t\ r}\ (\mu)
$$

Note that $\longrightarrow_{ex}$ recovers the ξ-rule. However, the resulting relation is still *weak* in the sense of Definition 5 on locally-closed terms, because of the β-rule enforcing that λ-abstractions only bind local variables, and that only locally-closed (i.e. weak) redexes are contracted. The two relations are equivalent on locally-closed terms (Theorem 13.) The proof is adapted from [13] (Proposition 4.6).

▶ **Lemma 11.** *If* $t \longrightarrow_{ex} s$*, then there exist a term* C *with free global variable* x *and* locally-closed *terms* a, b *such that* $a \longrightarrow_{w} b$ *and* $C\langle a/x\rangle \equiv t, C\langle b/x\rangle \equiv s$*.*

Proof. By induction on the reduction proof. We consider the two important cases

- Case (β). Then both terms are locally-closed, thus the contraction is valid in $\longrightarrow_{w}$. We take them as our a and b, and put $C \equiv x^G$ for x fresh;
- Case (ξ). Given $\lambda y.t \longrightarrow_{ex} \lambda y.s$, by induction hypothesis on $t \longrightarrow_{ex} s$ we have C', $a \longrightarrow_{w} b$ and $C'\langle a/x\rangle \equiv t, C'\langle b/x\rangle \equiv s$. We put $C \equiv \lambda y.C'$ and conclude. ◀

▶ **Lemma 12.** *The following substitution rule is admissible*

$$
\frac{a \longrightarrow_{ex} b}{t\langle a/x\rangle \longrightarrow_{ex} t\langle b/x\rangle}
$$

Proof. By structural induction on t. ◀

▶ **Theorem 13.** $t \longrightarrow_{w} s \iff t \longrightarrow_{ex} s$ *for any locally-closed terms* t *and* s*.*

Proof. We consider each implication separately.

- ($\Longrightarrow$) By observing that every rule in $\longrightarrow_w$ is admissible in $\longrightarrow_{ex}$.
- ($\Longleftarrow$) By Lemma 11, we get a term C and locally-closed terms a, b such that $a \longrightarrow_w b$ and $C\langle a/x\rangle \equiv t, C\langle b/x\rangle \equiv s$. We conclude by global variable substitution. ◀

Note that the equivalence in Theorem 13 extends to the reflexive transitive closures – since reduction does not affect free local variables – thus in particular the two relations give rise to the same set of normal forms for the same locally-closed term.

This result allows us to use the normalization function defined in this section to compute λ^{wk} normal forms, i.e. $\longrightarrow_w$-normal terms, provided such function computes $\longrightarrow_{ex}$-normal terms as we intended. We do not formally argue for this last point now. Rather, in the next sections we will adapt these ideas to the typed case, and define an "explicit" calculus based on $\longrightarrow_{ex}$ that is equivalent to the standard, "implicit" one. We will then prove the explicit calculus normalizing, and transporting along the equivalence will provide us with a normalization proof for the implicit calculus.

4 Typed weak λ-reduction

4.1 Simply-typed weak λ-calculus: λ^{wk}

We now define λ^{wk}, a simply-typed λ-calculus with weak λ-conversion judgments, that we aim to prove normalizing on well-typed terms. We use the same two-variable syntax as the previous Section. Types Ty and contexts Ctxt are defined as follows

$$\mathsf{Ty} \ni A, B ::= A \Rightarrow B \qquad\qquad \mathsf{Ctxt} \ni \Gamma ::= \cdot \mid \Gamma, x : A$$

Here we write $\Gamma \ni x : A$ for the predicate that is true when $x : A$ is included in Γ. We will assume that variables in contexts have unique names. The type system is shown in Figure 1. Even though we only care about locally-closed terms, we formulate the typing judgments in a slightly more general way, that considers arbitrary, not necessarily locally-closed terms. The judgments are of the form $\Gamma; \Delta \vdash t : A$, with a double context assigning a type to, respectively, "global" and "local" assumptions. We thus call the two contexts *global* and *local*. A well-typed term $\Gamma \vdash t : A$ of λ^{wk} is one where there are no free local variables, or equivalently, one that is typeable under an empty "local" context. That is, $\Gamma \vdash t : A :\equiv \Gamma; \cdot \vdash t : A$. Typed weak λ-conversion judgments are given by the inductive relation $\_ \vdash \_ \sim \_ : \_$, and essentially provide a formulation of the relation in Definition 9 with typed equality judgments.

4.2 Explicit weak λ-calculus: λ^{ex}

We now define a type system for λ^{ex}, with explicit weak λ-reduction (Definition 10) as equality judgments. The motivation for this system is the same behind λ^{ex} in the untyped case, namely to avoid the issues of λ^{wk} in specifying a normalization algorithm and proving it correct (see Section 2.1.) We will establish the following results:

- λ^{wk} and λ^{ex} are equivalent on locally-closed terms;
- λ^{ex} is normalizing on arbitrary (possibly non-locally-closed) well-typed terms.

Since every well-typed λ^{wk}-term is locally-closed, these two results imply what we ultimately seek to prove, namely normalization for λ^{wk} (Section 5.4). The advantage is that the actual normalization proof is carried out on λ^{ex}, which plays better with already-known proof methods that assume the ξ-rule, such as Normalization by Evaluation (as used, for example, in [11, 16].)

$\boxed{\_;\_ \vdash \_ : \_}$

$$\frac{\Gamma \ni x : A}{\Gamma;\Delta \vdash x^G : A} \qquad \frac{\Delta \ni x : A}{\Gamma;\Delta \vdash x^L : A} \qquad \frac{\Gamma;\Delta, x : A \vdash t : B}{\Gamma;\Delta \vdash \lambda x.t : A \Rightarrow B} \qquad \frac{\Gamma;\Delta \vdash t : A \Rightarrow B \quad \Gamma;\Delta \vdash s : A}{\Gamma;\Delta \vdash t\, s : B}$$

$\boxed{\_ \vdash \_ \sim \_ : \_}$

$$\frac{\Gamma; x : A \vdash t : B \qquad \Gamma \vdash s : A}{\Gamma \vdash (\lambda x.t)\, s \sim t[s/x] : B}\ (\beta) \qquad \frac{\Gamma, x : A; \cdot \vdash t : B \qquad \Gamma \vdash a \sim b : A}{\Gamma \vdash t\langle a/x\rangle \sim t\langle b/x\rangle : A}\ (\sigma)$$

$$\frac{\Gamma \vdash t : A}{\Gamma \vdash t \sim t : A} \qquad \frac{\Gamma \vdash t \sim s : A}{\Gamma \vdash s \sim t : A} \qquad \frac{\Gamma \vdash t \sim s : A \qquad \Gamma \vdash s \sim r : A}{\Gamma \vdash t \sim r : A}$$

Figure 1 Simply-typed λ-calculus with weak λ-reduction.

$\boxed{\_;\_ \vdash \_ \longrightarrow \_ : \_}$

$$\frac{\Gamma; x : A \vdash t : B \qquad \Gamma; \cdot \vdash s : A}{\Gamma;\Delta \vdash (\lambda x.t)s \longrightarrow t[s/x] : B}\ (\beta) \qquad \frac{\Gamma;\Delta, x : A \vdash t \longrightarrow s : B}{\Gamma;\Delta \vdash \lambda x.t \longrightarrow \lambda x.s : A \Rightarrow B}\ (\xi)$$

$$\frac{\Gamma;\Delta \vdash s : A}{\Gamma;\Delta \vdash t \longrightarrow t' : A \Rightarrow B} \qquad \frac{\Gamma;\Delta \vdash t : A \Rightarrow B}{\Gamma;\Delta \vdash s \longrightarrow s' : A} \qquad \frac{}{\Gamma;\Delta \vdash ts \longrightarrow t's : B} \qquad \frac{}{\Gamma;\Delta \vdash ts \longrightarrow ts' : B}$$

$\boxed{\_;\_ \vdash \_ \sim \_ : \_}$

$$\frac{\Gamma;\Delta \vdash t \longrightarrow s : A}{\Gamma;\Delta \vdash t \sim s : A} \qquad \frac{\Gamma;\Delta \vdash t : A}{\Gamma;\Delta \vdash t \sim t : A} \qquad \frac{\Gamma;\Delta \vdash t \sim s : A}{\Gamma;\Delta \vdash s \sim t : A} \qquad \frac{\Gamma;\Delta \vdash t \sim s : A \quad \Gamma;\Delta \vdash s \sim r : A}{\Gamma;\Delta \vdash t \sim r : A}$$

Figure 2 Reduction and conversion judgments of λ^{ex}.

The raw syntax and the typing judgments of λ^{ex} are the same as λ^{wk}, with the difference that terms can now be well-typed under an arbitrary local context. Figure 2 shows the definition of typed equality judgments for λ^{ex}, written $\Gamma;\Delta \vdash t \sim s : A$, and again formulated on arbitrary two-variable terms. Note that we do not axiomatize conversion directly, but define it as the equivalence closure of typed one-step reduction $\Gamma;\Delta \vdash t \longrightarrow s : A$, for purely technical reasons. Similarly to Lemma 12, we prove substitution admissible:

▶ **Lemma 14.** *If* $\Gamma, x : A;\Delta \vdash t : B$ *and* $\Gamma;\cdot \vdash a \sim b : A$*, then* $\Gamma;\Delta \vdash t\langle a/x\rangle \sim t\langle b/x\rangle : B$.

Proof. By induction on the derivation of t. ◀

4.3 Equivalence between λ^{wk} and λ^{ex}

We now show that λ^{wk} and λ^{ex} are equivalent on locally-closed terms. $\Gamma;\cdot \vdash t : A \iff \Gamma \vdash t : A$ follows by definition, thus we are left to show that $\Gamma \vdash t \sim s : A \iff \Gamma;\cdot \vdash t \sim s : A$. This is essentially the typed version of Theorem 13, and it relies on an adaptation of Lemma 11 with a proof that the term extraction involved preserves typing.

▶ **Lemma 15.** *For all derivations of a typed reduction* $\Gamma; \Delta \vdash t \longrightarrow s : A$*, there exist terms* C, a, b*, a type* X*, and a fresh variable* x *such that* $\Gamma, x : X; \Delta \vdash C : A$ *is derivable,* $\Gamma \vdash a \sim b : X$ *is derivable, and* $C\langle a/x\rangle \equiv t$*,* $C\langle b/x\rangle \equiv s$*.*

Proof. By induction on the derivation of $t \longrightarrow s$. ◀

▶ **Theorem 16.** *For all* Γ, A, t, s*,* $\Gamma \vdash t \sim s : A \iff \Gamma; \cdot \vdash t \sim s : A$*.*

Proof. Just an adaptation of the proof of Theorem 13 to the typed case. ◀

5 Normalization by Evaluation

Normalization by Evaluation (NbE) is a semantic method to prove normalization for typed λ-calculi. It was originally employed by Martin-Löf, although not under this name, to give a proof of normalization for a weak, combinatory version of his Intuitionistic Type Theory [35]. The method was later applied to the $\lambda\beta\eta$ calculus by Berger and Schwichtenberg [11], as a model construction where the interpretation function is invertible by an operation called *reification*. The composition of interpretation and reification is normalization. An advantage of NbE is that it can be justified by semantic arguments like logical relations [2] or *glueing* [16], rather than cumbersome term rewriting techniques. NbE amounts to establishing the following properties of a normalization function nf:

- Completeness: if $\Gamma \vdash t \sim s : A$, then $nf(t) = nf(s)$;
- Soundness: if $\Gamma \vdash t : A$, then $\Gamma \vdash t \sim nf(t) : A$.

From these properties we get that convertibility is equivalent to syntactic identity of normal forms: $\Gamma \vdash t \sim s : A \iff nf(t) = nf(s)$. Since syntactic identity of normal forms is decidable, so is convertibility.

In the rest of this section we give an overview of a proof of untyped NbE for λ^{ex}. This variant of NbE was originally detailed in [5], and differs from type-directed NbE like that of [35] by its reliance on a normalization function that acts on untyped raw syntax alone.

One motivation for using untyped NbE here stems from our formalization work, which involved several substitution lemmas that we though were easier to carry out on raw syntax. Another reason has to do with our plan to extend this work to dependent types. Dependent well-typed syntaxes have been historically difficult to develop inside Type Theory itself [19]. Work on quotient inductive-inductive types [8] seems to offer a viable solution, although it was too recent to have been considered here.

We employ untyped NbE as described in [2]. Most of the proof replicates [2] quite closely, so we only highlight the parts that are specific to λ^{ex}, and direct the reader to the Agda formalization or the author's MSc Thesis [37] for the details.

5.1 Semantic domain and interpretation

NbE relies on an interpretation function $[\![\_]\!]$ from the syntax to a semantic domain D, given an environment ρ assigning meaning to the free variables of the input term. In the case of weak λ-reduction, we can just use syntactic normal forms as semantic values, with syntactic identity as equality in the model. Hence we put $\mathsf{D} \equiv \mathsf{Term}$, and take the (partial) normalization function from Section 3 as interpretation $[\![\_]\!]$. Part of the proof of normalization will be to show that this function is total on well-typed terms.

5.2 Completeness of NbE

Completeness of NbE amounts to showing that the interpretation of convertible terms yields equal semantic values. In our case, we need to show that judgmentally equal terms *do* have a normal form, and this is the same. We follow [2] and strengthen our syntactic model by defining appropriate *semantic types* $\mathcal{A} \in \mathcal{P}(\mathsf{D})$, that is subsets of normal terms closed under neutral values: $\mathsf{Ne} \subseteq \mathcal{A} \subseteq \mathsf{Nf}$. These sets play a similar role to *saturated sets*, or Tait's sets of computable terms [39]. Given semantic types $\mathcal{A}, \mathcal{B}$, we can form the semantic function space $\mathcal{A} \to \mathcal{B}$ of all normal forms f that map elements $a \in \mathcal{A}$ to elements $f \bullet a \in \mathcal{B}$.

We interpret syntactic types as expected, namely $[\![A \Rightarrow B]\!] = [\![A]\!] \to [\![B]\!]$. Similarly, we interpret contexts Γ as subsets of substitutions $[\![\Gamma]\!] \in \mathcal{P}(\mathsf{Subst})$, namely those that map assumptions $\Gamma \ni x : A$ to values in $[\![A]\!]$. We write $[\![t]\!]\rho \simeq [\![s]\!]\rho \in \mathcal{A}$ whenever t and s evaluate to the same normal form in $\mathcal{A}$. We then define semantic typing and conversion judgments:

$$\Gamma; \Delta \models t : A :\equiv \forall(\rho \in [\![\Delta]\!]),\ [\![t]\!]\rho \simeq [\![t]\!]\rho \in [\![T]\!]$$

$$\Gamma; \Delta \models t \sim s : A :\equiv \forall(\rho \in [\![\Delta]\!]),\ [\![t]\!]\rho \simeq [\![s]\!]\rho \in [\![A]\!]$$

▶ **Theorem 17.** *For all t, s, A, Γ, Δ,*
- *if $\Gamma; \Delta \vdash t : A$, then $\Gamma; \Delta \models t : A$;*
- *if $\Gamma; \Delta \vdash t \longrightarrow s : A$, then $\Gamma; \Delta \models t \sim s : A$;*
- *if $\Gamma; \Delta \vdash t \sim s : A$, then $\Gamma; \Delta \models t \sim s : A$.*

Proof. By induction on the derivations. ◀

▶ **Corollary 18** (Completeness of NbE). *For all Γ, A, t, s,*
1. *If $\Gamma; \Delta \vdash t : A$, then t has a normal form, namely $[\![t]\!]$;*
2. *If $\Gamma; \Delta \vdash t \sim s : A$, then t and s have the same normal form, i.e. $[\![t]\!] = [\![s]\!]$.*

Proof. Both points follow from Theorem 17. ◀

A consequence of completeness of NbE is that $[\![\_]\!]$ is total on well-typed terms. We write $[\![t]\!]$ for the interpretation/normalization function applied to the empty substitution.

5.3 Kripke logical relations and soundness of NbE

Soundness of NbE is the statement that well-typed terms are convertible to their normal form. To prove this, we rely on the definition of a *Kripke logical relation*. Logical relations are families of relations defined by induction on (syntactic) types. Kripke logical relations [28] are additionally indexed by a set of *worlds* together with an accessibility relation, in the sense of Kripke semantics. In our case, worlds are represented by contexts and substitutions. Our logical relation relates well-typed terms with semantic values, i.e. normal forms:

$$\begin{aligned}&\Theta; \Gamma \vdash M \circledR N : A \Rightarrow B :\equiv \\ &\quad M = \lambda x.t \ \wedge\ N = \lambda x.d \ \wedge \\ &\quad (\text{for all } \Theta; \Delta \vdash \sigma : \Gamma \text{ and } \Theta; \Delta \vdash s \circledR a : A,\ \text{then } \Theta; \Delta \vdash t[\sigma[s/x]] \circledR d[\sigma[a/x]] : B)\end{aligned}$$

where we write $\Theta; \Delta \vdash \sigma : \Gamma$ for substitutions σ mapping assumptions $\Gamma \ni x : A$ to terms $\Theta; \Delta \vdash t : A$. We can show that related objects are convertible.

▶ **Lemma 19.** *If $\Gamma; \Delta \vdash t \circledR a : T$ then $\Gamma; \Delta \vdash t \sim a : T$.*

Proof. By induction on T and on the logical relation. ◀

Note that the proof of Lemma 19 crucially depends on the ξ-rule. We will now prove that every well-typed term is logically related to its semantics. This result is generally called *fundamental lemma of logical relations*, and it is stated below in a generalized version, where terms are related up to some parallel substitutions assigning logically-related pairs of terms/values to free variables. We write $\Gamma; \Delta \vdash_s \sigma \mathrel{®} \rho : \nabla$ for parallel substitutions σ, ρ mapping assumptions in ∇.

▶ **Lemma 20.** *If* $\Theta; \Gamma \vdash t : T$ *and* $\Theta; \Delta \vdash_s \sigma \mathrel{®} \rho : \Gamma$*, then* $\Theta; \Delta \vdash t[\sigma] \mathrel{®} [\![t]\!]\, \rho : T$.

Proof. By induction on the derivation of t. ◀

▶ **Theorem 21.** *[Soundness of NbE] If* $p \in \Gamma; \Delta \vdash t : A$*, then* $\Gamma; \Delta \vdash t \sim [\![t]\!] : A$.

Proof. By completeness of NbE, Lemma 20, and Lemma 19. ◀

As a consequence of NbE we get the following results (recall the beginning of Section 5):

▶ **Theorem 22** (Normalization of λ^{ex})**.** *If* $\Gamma; \Delta \vdash t : A$*, then* $\exists t'$ *normal s.t.* $\Gamma; \Delta \vdash t \sim t' : A$.

▶ **Corollary 23.** *Given* $\Gamma; \Delta \vdash t : A$*,* $\Gamma; \Delta \vdash s : A$*, the judgment* $\Gamma; \Delta \vdash t \sim s : A$ *is decidable.*

5.4 Normalization for λ^{wk}

From the equivalence result between λ^{wk} and λ^{ex} and normalization for the latter, we get

▶ **Theorem 24.** *[Normalization] If* $\Gamma \vdash t : A$*, then* $\exists t' \in \mathsf{Nf}$ *s.t.* $\Gamma \vdash t \sim t' : A$.

Proof. By Theorem 22 and Theorem 16. ◀

▶ **Corollary 25.** *If* $\Gamma \vdash t : A$ *and* $\Gamma \vdash s : A$*, then* $\Gamma \vdash t \sim s : A$ *is decidable.*

Proof. By Corollary 23, Theorem 16, and the fact that decidability respects logical equivalence. ◀

6 Conclusions

This article studies the notion of weak reduction originally due to Howard [25], and called here weak λ-reduction after [13]. In particular, it addresses the problem of defining an algorithmic procedure to compute normal forms of weak λ-reduction, and constructively proving the normalization theorem for a simply-typed λ-calculus equipped with this notion of reduction. The first part of this work includes a comparison of weak λ-reduction with other weak notions of reduction for the λ-calculus and combinatory logic. This comparison seems to reveal that these calculi are not, as currently developed, strong enough to compute weak λ-reduction normal forms. The solution proposed here relies instead on the definition of an "explicit" version of weak reduction that is equivalent to the original weak λ-reduction, and that facilitates the definition a normalization algorithm and the reasoning about its correctness. As far as we are aware, this provides the first detailed specification of a normalization algorithm for weak λ-reduction, and a proof of its correctness in the typed case. Our work has been fully formalized in the Agda proof assistant [2].

[2] The formalization can be found at `https://github.com/fsestini/nbe-weak-stlc`

6.1 Related work

Weak λ-reduction seems to have received limited attention in the literature, with some exceptions [25, 13] already mentioned in the previous sections. An early version of Martin-Löf Type Theory [35] had a primitive substitution rule and no ξ-rule. The author argues in [34] that a rule like ξ is unacceptable because stronger than what is intuitively and informally understood as definitional equality.

Weak λ-reduction is also employed in the Minimalist Foundation, a two-level foundation for constructive mathematics in the style of Martin-Löf Type Theory ideated by Sambin and Maietti in [33], and completed by Maietti to a formal system in [32]. There, the ξ-rule is rejected because it makes it easy to give a Kleene realizability interpretation for the theory, a property of ξ-free systems that was already noted in [34]. The realizability model is then used to show consistency with the formal Church Thesis and the Axiom of Choice [27].

Kesner et al. [29, 30] study weak-head reduction in the context of intersection types and call-by-need reduction strategies. Hyland and Ong [26] use weak reduction to construct a PCA of strongly-normalizing λ-terms as a basis for a general method to prove strong normalization for various type theories. The notion of equality in the PCA is a weak reduction relation similar to weak λ-reduction, that only contracts closed redexes. The same kind of restricted reduction is also employed in [22]. In [6], Akama introduces a translation from λ-terms to combinators, so that a term is strongly-normalizing under β-reduction if and only if its translation is strongly-normalizing under the weak conversion of combinatory logic.

The ideas presented in this article are exposed in more detail in the author's (unpublished) Master's Thesis [37], where normalization is proved for System T rather than simple types. The thesis also contains an analysis of the problem for systems with dependent types, and a proof of NbE for a version of Martin-Löf Type Theory with one universe and weak explicit substitutions. The author later discovered that a similar system had also been developed by Barras et al. [10, 15].

The proofs of normalization shown in this work are based on Normalization by Evaluation. NbE was first employed by Martin-Löf in [35] for his combinatory theory, although not under this name. The method was later rediscovered in [11] in the context of the simply-typed λ-calculus with η equality. Coquand and Dybjer [16] apply NbE for typed combinatory logic and two weak λ-calculi, using a model construction inspired by the categorical notion of *glueing*. Untyped NbE, originally developed in [5], is employed here following [2].

6.2 Future work

We would like to study further the connection between weak λ-reduction, weak explicit substitutions, and combinatory logic. In particular, we conjecture that weak explicit substitutions can be shown to simulate weak λ-reduction, given suitable translation functions between terms of the two calculi that we have sketched but not yet proved correct.

Another direction for the future is the extension of this work to weak systems with dependent types, most notably the intensional level of the Minimalist Foundation. Past attempts seem to suggest that the method exposed here does not scale well to dependent types with typed equality judgments. However, it does if the calculus is based on an untyped conversion relation, like the one considered in [3]. Thus, a solution could be to first prove that the weak Type Theory of interest, defined with typed equality judgments, is equivalent to its formulation based on untyped conversion, possibly using results from [38]. A different approach could be to show that the type theory with weak explicit substitutions in [37] is equivalent to the one with implicit substitutions.

References

1 M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. *Journal of Functional Programming*, 1(4):375–416, October 1991.

2 A. Abel. *Normalization by Evaluation, Dependent Types and Impredicativity.* Habilitation thesis, Institut für Informatik, Ludwig-Maximilians-Universität München, 2013.

3 Andreas Abel, Klaus Aehlig, and Peter Dybjer. Normalization by Evaluation for Martin-Löf Type Theory with One Universe. *Electronic Notes in Theoretical Computer Science*, 173:17–39, April 2007.

4 Samson Abramsky. The lazy lambda calculus. In David A. Turner, editor, *Research Topics in Functional Programming*, pages 65–116. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1990.

5 Klaus Aehlig and Felix Joachimski. Operational aspects of untyped Normalisation by Evaluation. *Mathematical Structures in Computer Science*, 14(4):587–611, 2004.

6 Yohji Akama. A λ-to-CL translation for strong normalization. In Philippe de Groote and J. Roger Hindley, editors, *Typed Lambda Calculi and Applications*, pages 1–10, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

7 Thorsten Altenkirch and James Chapman. Big-step Normalisation. *J. Funct. Program.*, 19(3–4):311–333, July 2009.

8 Thorsten Altenkirch and Ambrus Kaposi. Type Theory in Type Theory Using Quotient Inductive Types. *SIGPLAN Not.*, 51(1):18–29, January 2016. doi:10.1145/2914770.2837638.

9 Hendrik Pieter Barendregt. *The lambda calculus: its syntax and semantics.* North-Holland, 1984.

10 Bruno Barras, Thierry Coquand, and Simon Huber. A generalization of the Takeuti–Gandy interpretation. *Mathematical Structures in Computer Science*, 25(5):1071–1099, 2015. doi:10.1017/S0960129514000504.

11 U. Berger and H. Schwichtenberg. An inverse of the evaluation functional for typed lambda-calculus. In *[1991] Proceedings Sixth Annual IEEE Symposium on Logic in Computer Science*, page 203–211, July 1991.

12 Ana Bove, Peter Dybjer, and Ulf Norell. A Brief Overview of Agda - A Functional Language with Dependent Types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17-20, 2009. Proceedings*, pages 73–78, 2009. doi:10.1007/978-3-642-03359-9_6.

13 Naim Çağman and J.Roger Hindley. Combinatory weak reduction in lambda calculus. *Theoretical Computer Science*, 198(1):239–247, 1998.

14 Arthur Charguéraud. The Locally Nameless Representation. *Journal of Automated Reasoning*, 49(3):363–408, October 2012.

15 Thierry Coquand. Weak Type Theory (unpublished note). URL: http://www.cse.chalmers.se/~coquand/wmltt.pdf.

16 Thierry Coquand and Peter Dybjer. Intuitionistic Model Constructions and Normalization Proofs. *Preliminary Proceedings of the 1993 TYPES Workshop, Nijmegen*, 1993.

17 Pierre-Louis Curien, Thérèse Hardin, and Jean-Jacques Lévy. Confluence Properties of Weak and Strong Calculi of Explicit Substitutions. *J. ACM*, 43(2):362–397, March 1996.

18 H. B. Curry and R. Feys. Combinatory Logic. *Philosophische Rundschau*, 6(3/4):294, 1958.

19 Nils Anders Danielsson. A Formalisation of a Dependently Typed Language as an Inductive-Recursive Family. In Thorsten Altenkirch and Conor McBride, editors, *Types for Proofs and Programs*, pages 93–109, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

20 N. G de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381–392, January 1972.

21 Peter Dybjer and Denis Kuperberg. Formal Neighbourhoods, Combinatory Böhm Trees, and Untyped Normalization by Evaluation, 2008.

22 J.-Y. Girard. *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Thèse de doctorat d'État, Univerité Paris 7, 1972.

23 Robert Harper. *Practical Foundations for Programming Languages*. Cambridge University Press, 2012.

24 J. Roger Hindley and Jonathan P. Seldin. *Lambda-Calculus and Combinators: An Introduction*. Cambridge University Press, 2 edition, 2008.

25 W.A. Howard. Assignment of Ordinals to Terms for Primitive Recursive Functionals of Finite Type. In *Studies in Logic and the Foundations of Mathematics*. Elsevier, 1970.

26 J.M.E. Hyland and C. h. L. Ong. Modified Realizability Toposes and Strong Normalization Proofs (Extended Abstract). In *Typed Lambda Calculi and Applications, LNCS 664*, pages 179–194. Springer-Verlag, 1993.

27 Hajime Ishihara, Maria Emilia Maietti, Samuele Maschio, and Thomas Streicher. Consistency of the intensional level of the Minimalist Foundation with Church's thesis and axiom of choice. *Archive for Mathematical Logic*, January 2018.

28 Achim Jung and Jerzy Tiuryn. *A New Characterization of Lambda Definability*. Springer, 1993.

29 Delia Kesner. Reasoning About Call-by-need by Means of Types. In Bart Jacobs and Christof Löding, editors, *Foundations of Software Science and Computation Structures*, pages 424–441, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

30 Delia Kesner, Alejandro Ríos, and Andrés Viso. Call-by-Need, Neededness and All That. In Christel Baier and Ugo Dal Lago, editors, *Foundations of Software Science and Computation Structures*, pages 241–257, Cham, 2018. Springer International Publishing.

31 Jean-Jacques Lévy and Luc Maranget. Explicit Substitutions and Programming Languages. In *Foundations of Software Technology and Theoretical Computer Science*, Lecture Notes in Computer Science, page 181–200. Springer, Berlin, Heidelberg, December 1999.

32 M.E. Maietti. A minimalist two-level foundation for constructive mathematics. *Annals of Pure and Applied Logic*, 160(3):319–354, 2009.

33 M.E. Maietti and G. Sambin. Toward a Minimalist Foundation for Constructive Mathematics. *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, 48, 2005.

34 Per Martin-Löf. About Models for Intuitionistic Type Theories and the Notion of Definitional Equality. *Studies in Logic and the Foundations of Mathematics*, 82, December 1975.

35 Per Martin-Löf. *An Intuitionistic Theory of Types: Predicative Part*, volume 80 of *Logic Colloquium '73*, page 73–118. Elsevier, January 1975.

36 C.-H. Luke Ong. Fully Abstract Models of the Lazy Lambda Calculus. In *FOCS*, 1988.

37 Filippo Sestini. Normalization by Evaluation for Typed Lambda-Calculi with Weak Conversions (MSc's Thesis, unpublished), 2018. URL: `http://www.cs.nott.ac.uk/~psxfs5/msc-thesis.pdf`.

38 Vincent Siles and Hugo Herbelin. Pure Type System conversion is always typable. *Journal of Functional Programming*, 22(2):153–180, 2012.

39 W. W. Tait. Intensional interpretations of functionals of finite type. *Journal of Symbolic Logic*, 32(2):198–212, 1967.

40 A. S. Troelstra and D. van Dalen. Constructivism in Mathematics, Volume 2. *Studia Logica*, 50(2):355–356, 1991.

Cubical Assemblies, a Univalent and Impredicative Universe and a Failure of Propositional Resizing

Taichi Uemura

University of Amsterdam, Amsterdam, The Netherlands
t.uemura@uva.nl

Abstract

We construct a model of cubical type theory with a univalent and impredicative universe in a category of cubical assemblies. We show that this impredicative universe in the cubical assembly model does not satisfy a form of propositional resizing.

2012 ACM Subject Classification Theory of computation → Type theory; Theory of computation → Denotational semantics

Keywords and phrases Cubical type theory, Realizability, Impredicative universe, Univalence, Propositional resizing

Digital Object Identifier 10.4230/LIPIcs.TYPES.2018.7

Funding This work is part of the research programme "The Computational Content of Homotopy Type Theory" with project number 613.001.602, which is financed by the Netherlands Organisation for Scientific Research (NWO).

Acknowledgements I would like to thank Benno van den Berg, Martijn den Besten and Andrew Swan for helpful discussions and comments, and Bas Spitters, Steve Awodey and the anonymous reviewer for their comments, questions and suggestions.

1 Introduction

Homotopy type theory [33] is an extension of Martin-Löf's dependent type theory [29] with homotopy-theoretic ideas. The most important features are Voevodsky's *univalence axiom* and *higher inductive types* which provide a novel synthetic way of proving theorems of abstract homotopy theory and formalizing mathematics in computer proof assistants [4].

Ordinary homotopy type theory [33] uses a cumulative hierarchy of universes

$$\mathcal{U}_0 : \mathcal{U}_1 : \mathcal{U}_2 : \dots,$$

but there is another choice of universes: one *impredicative* universe in the style of the Calculus of Constructions [13]. Here we say a universe $\mathcal{U}$ is impredicative if it is closed under dependent products along any type family: for any type A and function $B : A \to \mathcal{U}$, the dependent product $\prod_{x:A} B(x)$ belongs to $\mathcal{U}$. An interesting use of such an impredicative universe in homotopy type theory is the *impredicative encoding* of higher inductive types, proposed by Shulman [35], as well as ordinary inductive types in polymorphic type theory [19]. For instance, the unit circle $\mathbb{S}^1$ is encoded as $\prod_{X:\mathcal{U}} \prod_{x:X} x = x \to X$ which has a base point and a loop on the point and satisfies the recursion principle in the sense of the HoTT book [33, Chapter 6]. Although the impredicative encoding of a higher inductive type does not satisfy the induction principle in general, some truncated higher inductive types have refinements of the encodings satisfying the induction principle [36, 2].

In this paper we construct a model of type theory with a univalent and impredicative universe to prove the consistency of that type theory. Impredicative universes are modeled in the category of *assemblies* or *ω-sets* [28, 32], while univalent universes are modeled in the categories of groupoids [21], simplicial sets [26] and cubical sets [5, 6]. Therefore, in order to

construct a univalent and impredicative universe, it is natural to combine them and construct a model of type theory in the category of internal groupoids, simplicial or cubical objects in the category of assemblies. There has been an earlier attempt to obtain a univalent and impredicative universe by Stekelenburg [38] who took a simplicial approach. A difficulty with this approach is that the category of assemblies does not satisfy the axiom of choice or law of excluded middle, so it becomes harder to obtain a model structure on the category of simplicial objects. Another approach is taken by van den Berg [43] using groupoid-like objects, but his model has a dimension restriction. Our choice is the cubical objects in the category of assemblies, which we will call *cubical assemblies*. Since the model in cubical sets [5, 10] is expressed, informally, in a constructive metalogic, one would expect that their construction can be translated into the internal language of the category of assemblies. A similar approach is taken by Awodey, Frey and Hofstra [1, 15].

Instead of a model of homotopy type theory itself, we construct a model of a variant of *cubical type theory* [10] in which the univalence axiom is provable. Orton and Pitts [30] gave a sufficient condition for modeling cubical type theory without universes of fibrant types in an elementary topos equipped with an interval object $\mathbb{I}$. Although the category of cubical assemblies is not an elementary topos, most of their proofs work in our setting because they use a dependent type theory as an internal language of a topos and the category of cubical assemblies is rich enough to interpret the type theory. For construction of the universe of fibrant types, we can use the right adjoint to the exponential functor $(\mathbb{I} \to -)$ in the same way as Licata, Orton, Pitts and Spitters [27].

Voevodsky [45] has proposed the *propositional resizing axiom* [33, Section 3.5] which implies that every homotopy proposition is equivalent to some homotopy proposition in the smallest universe. The propositional resizing axiom can be seen as a form of impredicativity for homotopy propositions. Since the universe in the cubical assembly model is impredicative, one might expect that the cubical assembly model satisfies the propositional resizing axiom. Indeed, for a homotopy proposition A, we have an approximation A^* of A by a homotopy proposition in $\mathcal{U}$ defined as

$$A^* := \prod_{X:\mathsf{hProp}} (A \to X) \to X,$$

where hProp is the universe of homotopy propositions in $\mathcal{U}$, and A is equivalent to some homotopy proposition in $\mathcal{U}$ if and only if the function $\lambda a X h.ha : A \to A^*$ is an equivalence. However, the propositional resizing axiom fails in the cubical assembly model. We construct a homotopy proposition A such that the function $A \to A^*$ is not an equivalence.

We begin Section 2 by formulating the axioms for modeling cubical type theory given by Orton and Pitts [30, 31] in a weaker setting. In Section 3 we describe how to construct a model of cubical type theory under those axioms. In Section 4 we give a sufficient condition for presheaf models to satisfy those axioms. As an example of presheaf model we construct a model of cubical type theory in cubical assemblies in Section 5, and show that the cubical assembly model does not satisfy the propositional resizing axiom.

2 The Orton-Pitts Axioms

We will work in a model $\mathcal{E}$ of dependent type theory with

- dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts and propositional truncation;
- a constant type $\vdash \mathbb{I}$, called an *interval*, with two constants $\vdash 0 : \mathbb{I}$ and $\vdash 1 : \mathbb{I}$ called *end-points* and two operators $i, j : \mathbb{I} \vdash i \sqcap j : \mathbb{I}$ and $i, j : \mathbb{I} \vdash i \sqcup j : \mathbb{I}$ called *connections*;

1. $\neg(0 = 1)$
2. $\forall_{i:\mathbb{I}} 0 \sqcap i = i \sqcap 0 = 0 \wedge 1 \sqcap i = i \sqcap 1 = i$
3. $\forall_{i:\mathbb{I}} 0 \sqcup i = i \sqcup 0 = i \wedge 1 \sqcup i = i \sqcup 1 = 1$
4. $i : \mathbb{I} \vdash i = 0 : \mathsf{Cof}$
5. $i : \mathbb{I} \vdash i = 1 : \mathsf{Cof}$
6. $\varphi, \psi : \mathsf{Cof} \vdash \varphi \vee \psi : \mathsf{Cof}$
7. $\varphi : \mathsf{Cof}, \psi : \varphi \to \mathsf{Cof} \vdash \sum_{u:\varphi} \psi u : \mathsf{Cof}$
8. $\varphi : \mathbb{I} \to \mathsf{Cof} \vdash \forall_{i:\mathbb{I}} \varphi i : \mathsf{Cof}$
9. $\forall_{\varphi,\psi:\mathsf{Cof}}(\varphi \leftrightarrow \psi) \to (\varphi = \psi)$
10. $\varphi : \mathsf{Cof}, A : \varphi \to \mathcal{U}, B : \mathcal{U}, f : \prod_{u:\varphi} Au \cong B \vdash \mathsf{iea}(\varphi, f) : \sum_{\bar{A}:\mathcal{U}} \{\bar{f} : \bar{A} \cong B \mid \forall_{u:\varphi}(Au, fu) = (\bar{A}, \bar{f})\}$

Figure 1 The Orton-Pitts Axioms.

- a dependent right adjoint to the exponential functor $(\mathbb{I} \to -) : \mathcal{E} \to \mathcal{E}$;
- a propositional universe $\vdash \mathsf{Cof}$ whose inhabitants are called *cofibrations*;
- an impredicative universe $\vdash \mathcal{U}$

satisfying the axioms listed in Figure 1. In the rest of the section we explain these conditions in more detail.

The dependent type theory we use is Martin-Löf's extensional type theory [29]. The notion of model of dependent type theory we have in mind is categories with families [14] equipped with certain algebraic operators corresponding to the type formers. A *category with families* $\mathcal{E}$ consists of:

- a category $\mathcal{E}$ of *contexts* with a terminal object denoted by $\cdot$;
- a presheaf $\Gamma \mapsto \mathcal{E}(\Gamma) : \mathcal{E}^{\mathrm{op}} \to \mathbf{Set}$ of *types*;
- a presheaf $(\Gamma, A) \mapsto \mathcal{E}(\Gamma \vdash A) : \mathbf{El}(\mathcal{E}(-))^{\mathrm{op}} \to \mathbf{Set}$ of *terms*, where $\mathbf{El}(P)$ is the category of elements for a presheaf P

such that, for any context $\Gamma \in \mathcal{E}$ and type $A \in \mathcal{E}(\Gamma)$, the presheaf

$$(\mathcal{E}/\Gamma)^{\mathrm{op}} \ni (\sigma : \Delta \to \Gamma) \mapsto \mathcal{E}(\Delta \vdash A\sigma) \in \mathbf{Set}$$

is representable, where $A\sigma$ denotes the element $P(\sigma)(A) \in P(\Delta)$ for a presheaf P, a morphism $\sigma : \Delta \to \Gamma$ and an element $A \in P(\Gamma)$. We assume that any category with families $\mathcal{E}$ has a choice of a representing object for this presheaf denoted by $\pi_A : \Gamma.A \to \Gamma$ and called the *context extension* of A. We also require that, for every context $\Gamma \in \mathcal{E}$, there exist types $C_0 \in \mathcal{E}(\cdot), C_1 \in \mathcal{E}(\cdot.C_0), \ldots, C_n \in \mathcal{E}(\cdot.C_0.\ldots.C_{n-1})$ and an isomorphism $\cdot.C_0.\ldots.C_n \cong \Gamma$. This means that, having dependent sum types, every context Γ can be thought of a closed type $\vdash \Gamma$. Type formers are modeled by algebraic operators. For example, to model dependent product types, $\mathcal{E}$ has an operator Π that carries triples (Γ, A, B) consisting of a context Γ and types $A \in \mathcal{E}(\Gamma)$ and $B \in \mathcal{E}(\Gamma.A)$ to types $\Pi(\Gamma, A, B) \in \mathcal{E}(\Gamma)$ and a bijection $l(\Gamma, A, B) : \mathcal{E}(\Gamma \vdash \Pi(\Gamma, A, B)) \cong \mathcal{E}(\Gamma.A \vdash B)$. These operators must be stable under base changes, that is, for any morphism $\sigma : \Delta \to \Gamma$, we have $\Pi(\Gamma, A, B)\sigma = \Pi(\Delta, A\sigma, B\sigma)$ and $l(\Gamma, A, B)\sigma = l(\Delta, A\sigma, B\sigma)$. All type-theoretic operations we introduce are required to be stable under base changes, unless otherwise stated. Note that there are alternative choices of notions of model of dependent type theory including categories with attributes [9] and split full comprehension categories [24]. Whichever model is chosen, we proceed entirely in its internal language.

In dependent type theory, a type $\Gamma \vdash \varphi$ is said to be a *proposition*, written $\Gamma \vdash \varphi\ \mathsf{Prop}$, if $\Gamma, u_1, u_2 : \varphi \vdash u_1 = u_2$ holds. For a proposition $\Gamma \vdash \varphi$, we say φ *holds* if there exists a (unique)

inhabitant of φ. For a type $\Gamma \vdash A$, its *propositional truncation* [3] is a proposition $\Gamma \vdash \|A\|$ equipped with a constructor $\Gamma, a : A \vdash |a| : \|A\|$ such that, for every proposition $\Gamma \vdash \varphi$, the function $\Gamma \vdash \lambda f a.f(|a|) : (\|A\| \to \varphi) \to (A \to \varphi)$ is an isomorphism. Propositions are closed under empty type, cartesian products and dependent products along arbitrary types, and we write $\bot, \top, \varphi \wedge \psi, \forall_{x:A}\varphi(x)$ for $\mathbf{0}, \mathbf{1}, \varphi \times \psi, \prod_{x:A}\varphi(x)$, respectively, when emphasizing that they are propositions. Also the identity type $\mathsf{Id}(A, a_0, a_1)$ is a proposition because it is extensional, and often written $a_0 = a_1$. The other logical operators are defined using propositional truncation as $\varphi \vee \psi := \|\varphi + \psi\|$ and $\exists_{x:A}\varphi(x) := \|\sum_{x:A}\psi(x)\|$. One can show that these logical operations satisfy the derivation rules of first-order intuitionistic logic. Moreover, the type theory admits subset comprehension defined as

$$\Gamma \vdash \{x : A \mid \varphi(x)\} := \sum_{x:A} \varphi(x)$$

for a proposition $\Gamma, x : A \vdash \varphi(x)$.

A finite coproduct $A + B$ is said to be *disjoint* if the inclusions $\mathsf{inl} : A \to A + B$ and $\mathsf{inr} : B \to A + B$ are monic and $\forall_{a:A}\forall_{b:B}\mathsf{inl}(a) \neq \mathsf{inr}(b)$ holds. A proposition $\Gamma \vdash \varphi$ is said to be *decidable* if $\Gamma \vdash \varphi \vee \neg\varphi$ holds. If the coproduct $\mathbf{2} := \mathbf{1} + \mathbf{1}$ of two copies of the unit type is disjoint, then it is a *decidable subobject classifier*: for every decidable proposition $\Gamma \vdash \varphi$, there exists a unique term $\Gamma \vdash b : \mathbf{2}$ such that $\Gamma \vdash \varphi \leftrightarrow (b = 1)$ holds. For readability we identify a boolean value $b : \mathbf{2}$ with the proposition $b = 1$.

For a functor $H : \mathcal{E} \to \mathcal{F}$ between the underlying categories of categories with families $\mathcal{E}$ and $\mathcal{F}$, a *dependent right adjoint* [7] to H consists of, for each context $\Gamma \in \mathcal{E}$ and type $A \in \mathcal{F}(H\Gamma)$, a type $G_\Gamma A \in \mathcal{E}(\Gamma)$ and an isomorphism $\varphi_A : \mathcal{F}(H\Gamma \vdash A) \cong \mathcal{E}(\Gamma \vdash G_\Gamma A)$ that are stable under reindexing in the sense that, for any morphism $\sigma : \Delta \to \Gamma$, we have $(G_\Gamma A)\sigma = G_\Delta(A\sigma)$ and $(\varphi_A a)\sigma = \varphi_{A\sigma}(a\sigma)$ for any $a \in \mathcal{F}(H\Gamma \vdash A)$. One can show that H preserves all colimits whenever it has a dependent right adjoint. As a consequence, assuming the exponential functor $(\mathbb{I} \to -)$ has a dependent right adjoint, the interval $\mathbb{I}$ is connected

$$\forall_{\varphi:\mathbb{I}\to\mathbf{2}}(\forall_{i:\mathbb{I}}\varphi i) \vee (\forall_{i:\mathbb{I}}\neg\varphi i),$$

which is postulated in [30] as an axiom.

A *universe* (à la Tarski) is a type $\vdash U$ equipped with a type $U \vdash \mathsf{el}_U$. We often omit the subscript $_U$ and simply write el for el_U if the universe is clear from the context. The universe U is said to be *propositional* if $U \vdash \mathsf{el}_U$ is a proposition. An *impredicative universe* is a universe U equipped with the following operations.

- A term $A : U, B : \mathsf{el}(A) \to U \vdash \sum^U(A, B) : U$ equipped with an isomorphism $A : U, B : \mathsf{el}(A) \to U \vdash e : \mathsf{el}(\sum^U(A, B)) \cong \sum_{x:\mathsf{el}(A)} \mathsf{el}(Bx)$.
- A term $A : U, a_0, a_1 : \mathsf{el}(A) \vdash \mathsf{Id}^U(A, a_0, a_1) : U$ equipped with an isomorphism $A : U, a_0, a_1 : \mathsf{el}(A) \vdash e : \mathsf{el}(\mathsf{Id}^U(A, a_0, a_1)) \cong (a_0 = a_1)$.
- For every type $\Gamma \vdash A$, a term $\Gamma, B : \mathsf{el}(A) \to U \vdash \prod^U(A, B) : U$ equipped with an isomorphism $\Gamma, B : \mathsf{el}(A) \to U \vdash e : \mathsf{el}(\prod^U(A, B)) \cong \prod_{x:A} \mathsf{el}(Bx)$.

One might want to require that $\mathsf{el}(\sum^U(A, B))$ is equal to $\sum_{x:\mathsf{el}(A)} \mathsf{el}(Bx)$ on the nose rather than up to isomorphism, but in the category of assemblies described in Section 5, the impredicative universe of partial equivalence relations does not satisfy this equation. For this reason, the distinction between terms $A : U$ and types $\mathsf{el}(A)$ is necessary, but for readability we often identify a term $A : U$ with the type $\mathsf{el}(A)$. For example, in Axiom 10 some el's should be inserted formally. Also Axiom 6 formally means that there exists a term $\varphi, \psi : \mathsf{Cof} \vdash \vee^{\mathsf{Cof}}(\varphi, \psi) : \mathsf{Cof}$ such that $\varphi, \psi : \mathsf{Cof} \vdash \mathsf{el}(\vee^{\mathsf{Cof}}(\varphi, \psi)) \leftrightarrow (\mathsf{el}(\varphi) \vee \mathsf{el}(\psi))$ holds.

Almost all the axioms in Figure 1 are direct translations of those in [30, 31]. Strictly speaking, Axioms 4 to 8 are part of structures rather than axioms in our setting, because Cof is no longer a subobject of the subobject classifier. Also Axiom 10, called the *isomorphism extension axiom*, is part of structures. As already mentioned, the connectedness of the interval $\mathbb{I}$ follows from the existence of the right adjoint to the exponential functor $(\mathbb{I} \to -)$. We need Axiom 9, which asserts the extensionality of the propositional universe Cof, for fibration structures on identity types. This axiom trivially holds in case that Cof is a subobject of the subobject classifier in an elementary topos. We also note that Cof is closed under $\bot$, $\top$ and $\wedge$ using Axioms 1, 5 and 7.

3 Modeling Cubical Type Theory

We describe how to construct a model of a variant of cubical type theory in our setting following Orton and Pitts [30]. Throughout the section $\mathcal{E}$ will be a model of dependent type theory satisfying the conditions explained in Section 2. Type-theoretic notations in this section are understood in the internal language of $\mathcal{E}$.

Cubical type theory is an extension of dependent type theory with an *interval object* [10, Section 3], the *face lattice* [10, Section 4.1], *systems* [10, Section 4.2], *composition operations* [10, Section 4.3] and the *gluing operation* [10, Section 6]. It also has several type formers including dependent product types, dependent sum types, *path types* [10, Section 3] and, optionally, *identity types* [10, Section 9.1]. We make some modifications to the original cubical type theory [10] in the same way as Orton and Pitts [30]. Major differences are as follows.

1. In [10] the interval object $\mathbb{I}$ is a de Morgan algebra, while we only require that $\mathbb{I}$ is a path connection algebra.
2. Due to the lack of de Morgan involution, we need composition operations in both directions "from 0 to 1" and "from 1 to 0".

In this section we will construct from $\mathcal{E}$ a new model of dependent type theory $\mathcal{E}^F$ that supports all operations of cubical type theory.

3.1 The Face Lattice and Systems

The *face lattice* [10, Section 4.1] is modeled by the propositional universe Cof. Note that in [10] quantification $\forall_{i:\mathbb{I}}\varphi$ is not part of syntax and written as a disjunction of irreducible elements, and plays a crucial role for defining composition operation for gluing. Since Cof need not admit quantifier elimination, we explicitly require Axiom 8.

We use the following operation for modeling *systems* [10, Section 4.2] which allows one to amalgamate compatible partial functions.

▶ **Proposition 1.** *One can derive an operation*

$$\frac{\begin{array}{c}\Gamma \vdash A \qquad \Gamma \vdash \varphi_i\ \mathsf{Prop} \qquad \Gamma, u_i : \varphi_i \vdash a_i(u_i) : A \\ \Gamma, u : \varphi_i, u' : \varphi_j \vdash a_i(u) = a_j(u') \qquad (i \text{ and } j \text{ run over } \{1, \dots, n\})\end{array}}{\Gamma \vdash [(u_1 : \varphi_1) \mapsto a_1(u_1), \dots, (u_n : \varphi_n) \mapsto a_n(u_n)] : \varphi_1 \vee \dots \vee \varphi_n \to A}$$

such that $\Gamma, v : \varphi_i \vdash [(u_1 : \varphi_1) \mapsto a_1(u_1), \dots, (u_n : \varphi_n) \mapsto a_n(u_n)]v = a_i(v)$ *for* $i = 1, \dots, n$.

Proof. Let B denote the union of images of a_i's:

$$\Gamma \vdash B := \{a : A \mid (\exists_{u_1:\varphi_1} a_1(u_1) = a) \vee \dots \vee (\exists_{u_n:\varphi_n} a_n(u_n) = a)\}.$$

Then $\Gamma \vdash B$ is a proposition because $\Gamma, u : \varphi_i, u' : \varphi_j \vdash a_i(u) = a_j(u')$ for all i and j. Hence the function $[a_1, \dots, a_n] : \varphi_1 + \dots + \varphi_n \to B$ induces a function $\|\varphi_1 + \dots + \varphi_n\| \to B$. ◀

3.2 Fibrations

We regard the type of Boolean values $\mathbf{2}$ as a subtype of the interval $\mathbb{I}$ via the end-point inclusion $[0,1] : \mathbf{2} \cong \mathbf{1} + \mathbf{1} \to \mathbb{I}$. We define a term $e : \mathbf{2} \vdash \bar{e} : \mathbf{2}$ as $\bar{0} = 1$ and $\bar{1} = 0$.

▶ **Definition 2.** *For a type* $\Gamma, i : \mathbb{I} \vdash A(i)$*, we define a type of* composition structures *as*

$$\Gamma \vdash \mathsf{Comp}^i(A(i)) := \prod_{e:\mathbf{2}} \prod_{\varphi:\mathsf{Cof}} \prod_{f:\varphi \to \prod_{i:\mathbb{I}} A(i)} \prod_{a:A(e)} (\forall_{u:\varphi} fue = a) \to \{a' : A(\bar{e}) \mid \forall_{u:\varphi} fu\bar{e} = a'\}.$$

In this notation, the variable i *is considered to be bound.*

▶ **Definition 3.** *For a type* $\gamma : \Gamma \vdash A(\gamma)$*, we define a type of* fibration structures *as*

$$\vdash \mathsf{Fib}(A) := \prod_{p:\mathbb{I}\to\Gamma} \mathsf{Comp}^i(A(pi)).$$

A fibration *is a type* $\Gamma \vdash A$ *equipped with a global section* $\vdash \alpha : \mathsf{Fib}(A)$*.*

For a fibration structure $\alpha : \mathsf{Fib}(A)$ on a type $\gamma : \Gamma \vdash A(\gamma)$ and a morphism $\sigma : \Delta \to \Gamma$, we define a fibration structure $\alpha\sigma : \mathsf{Fib}(A\sigma)$ on $\delta : \Delta \vdash A(\sigma(\delta))$ as

$$\alpha\sigma = \lambda p.\alpha(\sigma \circ p) : \prod_{p:\mathbb{I}\to\Delta} \mathsf{Comp}^i(A(\sigma(pi))).$$

Thus, for a fibration (A, α) on Γ, we have its *base change* $(A\sigma, \alpha\sigma)$ along a morphism $\sigma : \Delta \to \Gamma$. With this base change operation we get a model $\mathcal{E}^F$ of dependent type theory where

- the contexts are those of $\mathcal{E}$;
- the types over Γ are fibrations over Γ;
- the terms of a fibration $\Gamma \vdash A$ are terms of the underlying type $\Gamma \vdash A$ in $\mathcal{E}$

together with a forgetful map $\mathcal{E}^F \to \mathcal{E}$. In the same way as Orton and Pitts [30], one can show the following.

▶ **Theorem 4.** *The model of dependent type theory* $\mathcal{E}^F$ *supports:*

- *composition operations, path types and identity types; and*
- *dependent product types, dependent sum types, unit type and finite coproducts preserved by the forgetful map* $\mathcal{E}^F \to \mathcal{E}$*.*

We also introduce a class of objects that automatically carry fibration structures.

▶ **Definition 5.** *A type* $\vdash A$ *is said to be* discrete *if* $\forall_{f:\mathbb{I}\to A} \forall_{i:\mathbb{I}} fi = f0$ *holds.*

▶ **Proposition 6.** *If* $\vdash A$ *is a discrete type, then it has a fibration structure.*

Proof. Let $e : \mathbf{2}$, $\varphi : \mathsf{Cof}$, $f : \varphi \to \mathbb{I} \to A$ and $a : A$ such that $\forall_{u:\varphi} fue = a$. Then $a' := a : A$ satisfies $\forall_{u:\varphi} fu\bar{e} = a'$ by the discreteness. ◀

3.3 Path Types and Identity Types

For a type $\Gamma \vdash A$ and terms $\Gamma \vdash a_0 : A$ and $\Gamma \vdash a_1 : A$, we define the *path type* $\Gamma \vdash \mathsf{Path}(A, a_0, a_1)$ to be

$$\Gamma \vdash \{p : \mathbb{I} \to A \mid p0 = a_0 \wedge p1 = a_1\}.$$

We also define the *identity type* $\Gamma \vdash \mathsf{Id}(A, a_0, a_1)$ to be

$$\Gamma \vdash \sum_{p:\mathsf{Path}(A,a_0,a_1)} \{\varphi : \mathsf{Cof} \mid \varphi \to \forall_{i:\mathbb{I}} pi = a_0\}$$

which is a variant of Swan's construction [39]. Theorem 4 says that, if A has a fibration structure, then so do $\mathsf{Path}(A, a_0, a_1)$ and $\mathsf{Id}(A, a_0, a_1)$.

In the model $\mathcal{E}^F$, both path types and identity types admit the following introduction and elimination operations:

$$\frac{\Gamma \vdash a : A}{\Gamma \vdash \mathsf{refl}_a : P(A, a, a)} \quad P\text{-intro}$$

$$\frac{\begin{array}{c}\Gamma, x_0 : A, x_1 : A, z : P(A, x_0, x_1) \vdash C(z) \\ \Gamma, x : A \vdash c(x) : C(\mathsf{refl}_x) \qquad \Gamma \vdash a_0 : A \qquad \Gamma \vdash a_1 : A \qquad \Gamma \vdash p : P(A, a_0, a_1)\end{array}}{\Gamma \vdash \mathsf{ind}_{P(A)}(C, c, p) : C(p)} \quad P\text{-elim}$$

where P is either Path or Id. A difference between them is their computation rules. Identity types admit the *judgmental* computation rule like Martin-Löf's identity types:

$$\Gamma \vdash \mathsf{ind}_{\mathsf{Id}(A)}(C, c, \mathsf{refl}_a) = c(a)$$

for a term $\Gamma \vdash a : A$. On the other hand, path types only admit the *propositional* computation rule: for a term $\Gamma \vdash a : A$, one can find a term

$$\Gamma \vdash H(C, c, a) : \mathsf{Path}(C(a), \mathsf{ind}_{\mathsf{Path}(A)}(C, c, \mathsf{refl}_a), c(a)).$$

Therefore, when interpreting homotopy type theory, which is based on Martin-Löf's type theory, we use $\mathsf{Id}(A, a_0, a_1)$ rather than $\mathsf{Path}(A, a_0, a_1)$. However, it can be shown that $\mathsf{Id}(A, a_0, a_1)$ and $\mathsf{Path}(A, a_0, a_1)$ are equivalent, and thus we can replace $\mathsf{Id}(A, a_0, a_1)$ by simpler type $\mathsf{Path}(A, a_0, a_1)$ when analyzing the model $\mathcal{E}^F$ (see, for instance, the definition of homotopy proposition in Section 5.1).

3.4 Universes and Gluing

For a type $\gamma : \Gamma \vdash A(\gamma)$, a fibration structure on A corresponds to a term of the type $p : \mathbb{I} \to \Gamma \vdash \mathsf{C}(A)(p) := \mathsf{Comp}^i(A(pi))$. We define a type $\Gamma \vdash FA := \mathsf{C}(A)_{\mathbb{I}}$, using the dependent right adjoint $(-)_{\mathbb{I}}$ to the exponential functor $(\mathbb{I} \to -)$. By definition a morphism $\sigma : \Delta \to \sum_\Gamma FA$ corresponds to a pair (σ_0, α) consisting of a morphism $\sigma_0 : \Delta \to \Gamma$ and a fibration structure $\vdash \alpha : \prod_{p:\mathbb{I}\to\Delta} \mathsf{Comp}^i(A(\sigma_0(pi)))$.

Using this construction for the universe $\mathcal{U} \vdash \mathsf{el}_{\mathcal{U}}$, we have a new universe $\mathcal{U}^F := \sum_{\mathcal{U}} F(\mathsf{el})$ together with a fibration $(A, \alpha) : \mathcal{U}^F \vdash \mathsf{el}_{\mathcal{U}^F}(A, \alpha) := \mathsf{el}_{\mathcal{U}}(A)$. By definition $\mathcal{U}^F$ classifies fibrations whose underlying types belong to $\mathcal{U}$.

▶ **Theorem 7.** *The universe $\mathcal{U}^F$ is closed under dependent product types along arbitrary fibrations, dependent sum types and path types. If Cof belongs to $\mathcal{U}$, then $\mathcal{U}^F$ is closed under identity types.*

Proof. By Theorem 4, it suffices to show that $\mathcal{U}$ is closed under those type constructors, but this is clear by definition. ◀

We describe the *gluing operation* on the universe $\mathcal{U}^F$ following Orton and Pitts [30].

For a proposition $\Gamma \vdash \varphi$, types $\Gamma, u : \varphi \vdash A(u)$ and $\Gamma \vdash B$ and a function $\Gamma, u : \varphi \vdash f(u) : A(u) \to B$, we define a type $\mathsf{Glue}(\varphi, f)$ to be

$$\Gamma \vdash \mathsf{Glue}(\varphi, f) := \sum_{a : \prod_{u:\varphi} A(u)} \{b : B \mid \forall_{u:\varphi} f(u)(au) = b\}.$$

There is a canonical isomorphism $\Gamma, u : \varphi \vdash e(u) := \lambda(a, b).au : \mathsf{Glue}(\varphi, f) \cong A(u)$ with inverse $\lambda a.(\lambda v.a, f(u)a)$.

▶ **Proposition 8.** *For $\gamma : \Gamma \vdash \varphi(\gamma) : \mathsf{Cof}$, $\gamma : \Gamma, u : \varphi(\gamma) \vdash A(u)$, $\gamma : \Gamma \vdash B(\gamma)$ and $\gamma : \Gamma, u : \varphi(\gamma) \vdash f(u) : A(u) \to B$, if A and B are fibrations and f is an equivalence, then $\gamma : \Gamma \vdash \mathsf{Glue}(\varphi(\gamma), f)$ has a fibration structure preserved by the canonical isomorphism $\Gamma, u : \varphi \vdash e(u) : \mathsf{Glue}(\varphi, f) \cong A(u)$.*

Proof. The construction is similar to the definition of the composition operation for glue types [10, Section 6.2]. ◀

Since the universe $\mathcal{U}$ is closed under type formers used in the definition of $\mathsf{Glue}(\varphi, f)$, we get a term

$$\varphi : \mathsf{Cof}, A : \varphi \to \mathcal{U}, B : \mathcal{U}, f : \prod_{u:\varphi} A(u) \to B \vdash \mathsf{Glue}(\varphi, f) : \mathcal{U}$$

such that $\prod_{u:\varphi} \mathsf{Glue}(\varphi, f) \cong A(u)$. However, the gluing operation in cubical type theory [10, Section 6] requires that, assuming $u : \varphi$, $\mathsf{Glue}(\varphi, f)$ is equal to $A(u)$ on the nose rather than up to isomorphism. So we use Axiom 10 and get a term

$$\varphi : \mathsf{Cof}, A : \varphi \to \mathcal{U}, B : \mathcal{U}, f : \prod_{u:\varphi} A(u) \to B \vdash \mathsf{SGlue}(\varphi, f) : \mathcal{U}$$

such that $\mathsf{SGlue}(\varphi, f) \cong \mathsf{Glue}(\varphi, f)$ and $\forall_{u:\varphi} \mathsf{SGlue}(\varphi, f) = A(u)$. By Proposition 8 we also have a term

$$\varphi : \mathsf{Cof}, A : \varphi \to \mathcal{U}^F, B : \mathcal{U}^F, f : \prod_{u:\varphi} A(u) \simeq B \vdash \mathsf{SGlue}(\varphi, f) : \mathcal{U}^F$$

such that $\mathsf{SGlue}(\varphi, f) \cong \mathsf{Glue}(\varphi, f)$ and $\forall_{u:\varphi} \mathsf{SGlue}(\varphi, f) = A(u)$. Hence the universe $\mathcal{U}^F$ in the model $\mathcal{E}^F$ supports the gluing operation. The composition operation for universes is defined using the gluing operation [10, Section 7.1], so we have the following proposition.

▶ **Proposition 9.** *$\vdash \mathcal{U}^F$ has a fibration structure.*

Since the univalence axiom can be derived from the gluing operation [10, Section 7], we conclude that $\mathcal{U}^F$ is a univalent and impredicative universe in the model of cubical type theory $\mathcal{E}^F$.

4 Presheaf Models

In this section we give a sufficient condition for a presheaf category to satisfy the conditions in Section 2. We will work in a model $\mathcal{S}$ of dependent type theory with dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts and propositional truncation.

A *category* in $\mathcal{S}$ consists of:

- a type $\vdash \mathbf{C}_0$ of *objects*;
- a type $c_0, c_1 : \mathbf{C}_0 \vdash \mathbf{C}_1(c_0, c_1)$ of *morphisms*;
- a term $c : \mathbf{C}_0 \vdash \mathsf{id}_c : \mathbf{C}_1(c, c)$ called *identity*;
- a term $c_0, c_1, c_2 : \mathbf{C}_0, g : \mathbf{C}_1(c_1, c_2), f : \mathbf{C}_1(c_0, c_1) \vdash gf : \mathbf{C}_1(c_0, c_2)$ called *composition*

satisfying the standard axioms of category. We will simply write $\mathbf{C}$ and $\mathbf{C}(c_0, c_1)$ for $\mathbf{C}_0$ and $\mathbf{C}_1(c_0, c_1)$ respectively. The notions of functor and natural transformation in $\mathcal{S}$ are defined in the obvious way. For a category $\mathbf{C}$ in $\mathcal{S}$, a *presheaf* on $\mathbf{C}$ consists of:

- a type $c : \mathbf{C} \vdash A(c)$;
- a term $c_0, c_1 : \mathbf{C}, \sigma : \mathbf{C}(c_0, c_1), a : A(c_1) \vdash a\sigma : A(c_0)$ called *(right)* $\mathbf{C}$*-action*

satisfying $a\mathsf{id} = a$ and $a(\sigma\tau) = (a\sigma)\tau$. For presheaves A and B, a *morphism* $f : A \to B$ is a term $c : \mathbf{C}, a : A(c) \vdash f(a) : B(c)$ satisfying $c_0, c_1 : \mathbf{C}, \sigma : \mathbf{C}(c_0, c_1), a : A(c_1) \vdash f(a\sigma) = f(a)\sigma$. For a presheaf A, its *category of elements*, written $\mathbf{El}(A)$, is defined as

- $\vdash \mathbf{El}(A)_0 := \sum_{c:\mathbf{C}_0} A(c)$;
- $(c_0, a_0), (c_1, a_1) : \mathbf{El}(A)_0 \vdash \mathbf{El}(A)_1((c_0, a_0), (c_1, a_1)) := \{\sigma : \mathbf{C}_1(c_0, c_1) \mid a_1\sigma = a_0\}$.

There is a projection functor $\pi_A : \mathbf{El}(A) \to \mathbf{C}$.

For a category $\mathbf{C}$ in $\mathcal{S}$, we describe the presheaf model $\mathbf{PSh}(\mathbf{C})$ of dependent type theory. Contexts are interpreted as presheaves on $\mathbf{C}$. For a context Γ, types on Γ are interpreted as presheaves on $\mathbf{El}(\Gamma)$. For a type $\Gamma \vdash A$, terms of A are interpreted as sections of the projection $\pi_A : \mathbf{El}(A) \to \mathbf{El}(\Gamma)$. For a type $\Gamma \vdash A$, the context extension $\Gamma.A$ is interpreted as the presheaf $c : \mathbf{C} \vdash \sum_{\gamma:\Gamma(c)} A(c, \gamma)$. This construction is also used for dependent sum types. The dependent product for a type $\Gamma.A \vdash B$ is the presheaf

$$(c, \gamma) : \mathbf{El}(\Gamma) \vdash \{f : \prod_{c':\mathbf{C}} \prod_{\sigma:\mathbf{C}(c',c)} \prod_{a:A(c',\gamma\sigma)} B(c', a) \mid \forall_{c',c'':\mathbf{C}} \forall_{\sigma:\mathbf{C}(c',c)} \forall_{\tau:\mathbf{C}(c'',c')} \forall_{a:A(c',\gamma\sigma)} (fc'\sigma a)\tau = fc''(\sigma\tau)(a\tau)\}.$$

Extensional identity types, unit type, disjoint finite coproducts and propositional truncation are pointwise.

4.1 Lifting Universes

We describe the Hofmann-Streicher lifting of a universe [20]. Let $\mathbf{C}$ be a category in $\mathcal{S}$ and U a universe in $\mathcal{S}$. We define a universe $[\mathbf{C}^{\mathrm{op}}, U]$ in $\mathbf{PSh}(\mathbf{C})$ as follows. The universe U can be seen as a category whose type of objects is U and type of morphisms is $A, B : U \vdash \mathsf{el}_U(A) \to \mathsf{el}_U(B)$. For an object $c : \mathbf{C}$, we define $[\mathbf{C}^{\mathrm{op}}, U](c)$ to be the type of functors from $(\mathbf{C}/c)^{\mathrm{op}}$ to U. The $\mathbf{C}$-action on $[\mathbf{C}^{\mathrm{op}}, U]$ is given by precomposition. The type $[\mathbf{C}^{\mathrm{op}}, U] \vdash \mathsf{el}_{[\mathbf{C}^{\mathrm{op}},U]}$ in $\mathbf{PSh}(\mathbf{C})$ is defined as $(c, A) : \mathbf{El}([\mathbf{C}^{\mathrm{op}}, U]) \vdash \mathsf{el}_{[\mathbf{C}^{\mathrm{op}},U]}(c, A) := \mathsf{el}_U(A(\mathsf{id}_c))$.

It is easy to show that, if U is an impredicative universe, then dependent product types, dependent sum types and extensional identity types in U can be lifted to those in $[\mathbf{C}^{\mathrm{op}}, U]$ so that $[\mathbf{C}^{\mathrm{op}}, U]$ is an impredicative universe in $\mathbf{PSh}(\mathbf{C})$. If U is a propositional universe in $\mathcal{S}$, then $[\mathbf{C}^{\mathrm{op}}, U]$ is a propositional universe in $\mathbf{PSh}(\mathbf{C})$.

▶ **Proposition 10.** *Let $\mathcal{U}$ be an impredicative universe and* Cof *a propositional universe in $\mathcal{S}$. If they satisfy Axioms 6, 7, 9 and 10, then so do $[\mathbf{C}^{\mathrm{op}}, \mathcal{U}]$ and $[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$.*

Proof. We only check Axiom 10. The other axioms are easy to verify.

We have to define a term $\varphi : [\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}], A : \varphi \to [\mathbf{C}^{\mathrm{op}}, \mathcal{U}], B : [\mathbf{C}^{\mathrm{op}}, \mathcal{U}], f : \prod_{u:\varphi} Au \cong B \vdash (D(\varphi, f), g(\varphi, f)) : \sum_{\bar{A}:[\mathbf{C}^{\mathrm{op}},\mathcal{U}]} \{\bar{f} : \bar{A} \cong B \mid \forall_{u:\varphi}(Au, fu) = (\bar{A}, \bar{f})\}$ in $\mathbf{PSh}(\mathbf{C})$. It corresponds to a natural transformation that takes an object $c : \mathbf{C}$, functors $\varphi : (\mathbf{C}/c)^{\mathrm{op}} \to \mathsf{Cof}$, $A :$

$\mathbf{El}(\varphi)^{\mathrm{op}} \to \mathcal{U}$ and $B : (\mathbf{C}/c)^{\mathrm{op}} \to \mathcal{U}$ and an isomorphism $f : A \cong B\pi_\varphi$ of presheaves on $\mathbf{El}(\varphi)$ and returns a pair $(D(c,\varphi,f), g(c,\varphi,f))$ consisting of a functor $D(c,\varphi,f) : (\mathbf{C}/c)^{\mathrm{op}} \to \mathcal{U}$ and an isomorphism $g(c,\varphi,f) : A \cong B$ of presheaves on $(\mathbf{C}/c)^{\mathrm{op}}$ such that $D(c,\varphi,f)\pi_\varphi = A$ and $g(c,\varphi,f)\pi_\varphi = f$. Let $\sigma : \mathbf{C}(c',c)$ be a morphism. Then we have $\varphi(\sigma) : \mathsf{Cof}$, $\lambda u.A(\sigma,u) : \varphi(\sigma) \to \mathcal{U}$, $B(\sigma) : \mathcal{U}$ and an isomorphism $\lambda u.f(\sigma,u) : \prod_{u:\varphi(\sigma)} A(\sigma,u) \cong B(\sigma)$. By the isomorphism lifting on $\mathcal{U}$, we have $D(c,\varphi,f)(\sigma) : \mathcal{U}$ and an isomorphism $g(c,\varphi,f)(\sigma) : D(c,\varphi,f)(\sigma) \cong B(\sigma)$ such that $\forall_{u:\varphi(\sigma)}(A(\sigma,u), f(\sigma,u)) = (D(c,\varphi,f)(\sigma), g(c,\varphi,f)(\sigma))$. For the morphism part of the functor $D(c,\varphi,f)$, let $\tau : \mathbf{C}(c'',c')$ be another morphism. Then we define $\tau^* : D(c,\varphi,f)(\sigma) \to D(c,\varphi,f)(\sigma\tau)$ to be the composition

$$D(c,\varphi,f)(\sigma) \xrightarrow[\cong]{g(c,\varphi,f)(\sigma)} B(\sigma) \xrightarrow{\quad\tau^*\quad} B(\sigma\tau) \xrightarrow[\cong]{g(c,\varphi,f)(\sigma\tau)^{-1}} D(c,\varphi,f)(\sigma\tau).$$

By definition $g(c,\varphi,f)$ becomes a natural isomorphism and $(D(c,\varphi,f)\pi_\varphi, g(c,\varphi,f)\pi_\varphi) = (A,f)$. It is easy to see the naturality of $(c,\varphi,f) \mapsto (D(c,\varphi,f), g(c,\varphi,f))$. ◀

4.2 Intervals

Suppose a category $\mathbf{C}$ in $\mathcal{S}$ has finite products. A *path connection algebra* in $\mathbf{C}$ consists of an object $\mathbb{I} : \mathbf{C}$, morphisms $\delta_0, \delta_1 : \mathbf{C}(1,\mathbb{I})$ called *end-points* and morphisms $\mu_0, \mu_1 : \mathbf{C}(\mathbb{I}\times\mathbb{I}, \mathbb{I})$ called *connections* satisfying $\mu_e(\delta_e \times \mathbb{I}) = \mu_e(\mathbb{I} \times \delta_e) = \delta_e$ and $\mu_e(\delta_{\bar{e}} \times \mathbb{I}) = \mu_e(\mathbb{I} \times \delta_{\bar{e}}) = \mathsf{id}$ for $e \in \{0,1\}$.

For a path connection algebra $\mathbb{I}$ in $\mathbf{C}$, we have a representable presheaf $\mathbf{y}\mathbb{I}$ on $\mathbf{C}$. Since the Yoneda embedding is fully faithful and preserves finite products, $\mathbf{y}\mathbb{I}$ has end-points and connections satisfying Axioms 2 and 3. The interval $\mathbf{y}\mathbb{I}$ satisfies Axiom 1 if and only if $\forall_{c:\mathbf{C}}\delta_0!_c \neq \delta_1!_c$ holds, where $!_c : \mathbf{C}(c,1)$ is the unique morphism into the terminal object.

▶ **Proposition 11.** *Let* Cof *be a propositional universe in* $\mathcal{S}$ *and suppose that, for every pair of objects* $c, c' : \mathbf{C}$*, the equality predicate on* $\mathbf{C}(c,c')$ *belongs to* Cof*. Then, for every object* $c : \mathbf{C}$*, the equality predicate on* $\mathbf{y}c$ *belongs to* $[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$*. In particular,* $\mathbf{y}\mathbb{I}$ *and* $[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$ *in* $\mathbf{PSh}(\mathbf{C})$ *satisfy Axioms 4 and 5.*

Proof. Because equality on a presheaf is pointwise. ◀

▶ **Proposition 12.** *For a functor* $f : \mathbf{C} \to \mathbf{D}$ *between categories in* $\mathcal{S}$*, the precomposition functor* $f^* : \mathbf{PSh}(\mathbf{D}) \to \mathbf{PSh}(\mathbf{C})$ *has a dependent right adjoint* f_**.*

Proof. For a context Γ in $\mathbf{PSh}(\mathbf{D})$ and a type $f^*\Gamma \vdash A$ in $\mathbf{PSh}(\mathbf{C})$, the type $\Gamma \vdash f_*A$ is given by the presheaf $(d,\gamma) : \mathbf{El}(\Gamma) \vdash \lim_{(c,\sigma):(f\downarrow d)} A(c, \gamma\sigma)$. ◀

▶ **Proposition 13.** *Suppose that a category* $\mathbf{C}$ *in* $\mathcal{S}$ *has finite products. For an object* $c : \mathbf{C}$*, the exponential functor* $(\mathbf{y}c \to -) : \mathbf{PSh}(\mathbf{C}) \to \mathbf{PSh}(\mathbf{C})$ *is isomorphic to* $(- \times c)^*$*.*

Proof. $(\mathbf{y}c \to A)(c') \cong \mathbf{PSh}(\mathbf{C})(\mathbf{y}c' \times \mathbf{y}c, A) \cong \mathbf{PSh}(\mathbf{C})(\mathbf{y}(c' \times c), A) \cong A(c' \times c)$. ◀

Hence the exponential functor $(\mathbf{y}\mathbb{I} \to -)$ has a dependent right adjoint. Proposition 13 also implies Axiom 8 for the propositional universe $[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$. Explicitly, $\forall_{\mathbf{y}\mathbb{I}} : (- \times \mathbf{y}\mathbb{I})^*[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}] \to [\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$ is a natural transformation that carries a functor $\varphi : (\mathbf{C}/c \times \mathbb{I})^{\mathrm{op}} \to \mathsf{Cof}$ to $\lambda\sigma.\varphi(\sigma \times \mathbb{I}) : (\mathbf{C}/c)^{\mathrm{op}} \to \mathsf{Cof}$.

In summary, we have:

▶ **Theorem 14.** *Suppose:*

- $\mathcal{S}$ *is a model of dependent type theory with dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts and propositional truncation;*
- Cof *is a propositional universe and* $\mathcal{U}$ *is an impredicative universe satisfying Axioms 6, 7, 9 and 10;*
- $\mathbf{C}$ *is a category in* $\mathcal{S}$ *with finite products and the equality on* $\mathbf{C}(c, c')$ *belongs to* Cof *for every pair of objects* $c, c' : \mathbf{C}$*;*
- $\mathbb{I}$ *is a path connection algebra in* $\mathbf{C}$*;*
- $\mathbf{y}\mathbb{I}$ *satisfies Axiom 1.*

Then the presheaf model $\mathbf{PSh}(\mathbf{C})$ *together with propositional universe* $[\mathbf{C}^{\mathrm{op}}, \mathsf{Cof}]$*, impredicative universe* $[\mathbf{C}^{\mathrm{op}}, \mathcal{U}]$ *and interval* $\mathbf{y}\mathbb{I}$ *satisfies all the axioms in Figure 1.*

4.3 Decidable Subobject Classifier

An example of the propositional universe Cof in Theorem 14 is the decidable subobject classifier $\mathbf{2}$ which always satisfies Axioms 6, 7 and 9.

▶ **Proposition 15.** *In a model of dependent type theory with dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts and propositional truncation, any universe* $\mathcal{U}$ *satisfies Axiom 10 with* $\mathsf{Cof} = \mathbf{2}$*.*

Proof. Let $\varphi : \mathbf{2}, A : \varphi \to \mathcal{U}, B : \mathcal{U}, f : \prod_{u:\varphi} Au \cong B$. We define $\mathsf{iea}(\varphi, f)$ by case analysis on $\varphi : \mathbf{2}$ as $\mathsf{iea}(0, f) := (B, \mathsf{id})$ and $\mathsf{iea}(1, f) := (A*, f*)$ where $*$ is the unique element of a singleton type. ◀

4.4 Categories of Cubes

We present examples of internal categories $\mathbf{C}$ with a path connection algebra $\mathbb{I}$ satisfying the hypotheses of Theorem 14 with $\mathsf{Cof} = \mathbf{2}$. Obvious choices of $\mathbf{C}$ are the category of free de Morgan algebras [10] and various syntactic categories of the language $\{0, 1, \sqcap, \sqcup\}$ [8], but some inductive types and quotient types are required to construct these categories in dependent type theory. Although the motivating example of $\mathcal{S}$, the category of assemblies described in Section 5, has inductive types and finite colimits, quotients are not well-behaved in general and we need to be careful in using quotients. Instead, we give examples definable only using natural numbers.

Suppose $\mathcal{S}$ is a model of dependent type theory with dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts, propositional truncation and natural numbers. We define a type of finite types $n : \mathbb{N} \vdash \mathsf{Fin}_n$ to be $\mathsf{Fin}_n = \{k : \mathbb{N} \mid k < n\}$. We define a category $\mathbf{B}$ as follows. Its object of objects is $\mathbb{N}$. The morphisms $m \to n$ are functions $(\mathsf{Fin}_m \to \mathbf{2}) \to (\mathsf{Fin}_n \to \mathbf{2})$. In the category $\mathbf{B}$, the terminal object is $0 : \mathbb{N}$ and the product of m and n is $m + n$. One can show, by induction, that every $\mathbf{B}(m, n)$ has decidable equality. $\mathbf{B}$ has a path connection algebra $1 : \mathbb{N}$ together with end-points $0, 1 : (\mathsf{Fin}_0 \to \mathbf{2}) \to (\mathsf{Fin}_1 \to \mathbf{2})$ and connections $\mathsf{min}, \mathsf{max} : (\mathsf{Fin}_1 \to \mathbf{2}) \times (\mathsf{Fin}_1 \to \mathbf{2}) \to (\mathsf{Fin}_1 \to \mathbf{2})$. One can show that the category $\mathbf{B}$ satisfies the hypotheses of Theorem 14. Moreover, any subcategory of $\mathbf{B}$ that has the same finite products and contains the path connection algebra 1 satisfies the same condition. An example is the wide subcategory $\mathbf{B}_{\mathsf{ord}}$ of $\mathbf{B}$ where the morphisms are order-preserving functions $(\mathsf{Fin}_m \to \mathbf{2}) \to (\mathsf{Fin}_n \to \mathbf{2})$.

4.5 Constant and Codiscrete Presheaves

We show some properties of constant and codiscrete presheaves which will be used in Section 5. Let $\mathcal{S}$ be a model of dependent type theory satisfying the hypotheses of Theorem 14. For an object $A \in \mathcal{S}$, we define the *constant presheaf* ΔA to be $\Delta A(c) := A$ with the trivial $\mathbf{C}$-action.

▶ **Proposition 16.** *Every constant presheaf ΔA is discrete.*

Proof. For every $c : \mathbf{C}$, we have $(\mathbf{y}\mathbb{I} \to \Delta A)(c) \cong \Delta A(c \times \mathbb{I}) = A$ by Proposition 13. ◀

For a type $\Gamma \vdash A$ in $\mathcal{S}$, we define the *codiscrete presheaf* $\Delta\Gamma \vdash \nabla A$ to be $\nabla A(c, \gamma) := \mathbf{C}(1, c) \to A(\gamma)$ with composition as the $\mathbf{C}$-action.

▶ **Proposition 17.** *Suppose that* $\mathsf{Cof} = \mathbf{2}$*. Then for every type $\Gamma \vdash A$ in $\mathcal{S}$, the type $\Delta\Gamma \vdash \nabla A$ has a fibration structure.*

Proof. Since $\Delta\Gamma$ is discrete, it suffices to show that $\nabla A(\gamma)$ has a fibration structure for every $\gamma : \Gamma$. Thus we may assume that Γ is the empty context. We construct a term

$$\alpha : \prod_{e:\mathbf{2}} \prod_{\varphi:[\mathbf{C}^{\mathrm{op}},\mathbf{2}]} \prod_{f:\varphi\to\mathbb{I}\to\nabla A} \prod_{a:\nabla A} (\forall_{u:\varphi} fue = a) \to \{\bar{a} : \nabla A \mid \forall_{u:\varphi} fu\bar{e} = \bar{a}\}$$

in $\mathbf{PSh}(\mathbf{C})$. It corresponds to a natural transformation that takes an object $c : \mathbf{C}$, an element $e : \mathbf{2}$, a functor $\varphi : (\mathbf{C}/c)^{\mathrm{op}} \to \mathbf{2}$, a natural transformation $f : \int_{c' \in \mathbf{C}} (\sum_{\sigma:\mathbf{C}(c',c)} \varphi(\sigma)) \times \mathbf{C}(c', \mathbb{I}) \to \nabla A(c')$ and an element $a : \nabla A(c)$ such that $\forall_{c':\mathbf{C}} \forall_{\sigma:\mathbf{C}(c',c)} \forall_{u:\varphi(\sigma)} f(\sigma, u, e) = a\sigma$ and returns an element $\alpha(e, \varphi, f, a) : \nabla A(c)$ such that $\forall_{c':\mathbf{C}} \forall_{\sigma:\mathbf{C}(c',c)} \forall_{u:\varphi(\sigma)} f(\sigma, u, \bar{e}) = \alpha(e, \varphi, f, a)\sigma$. We define $\alpha(e, \varphi, f, a) : \mathbf{C}(1, c) \to A$ as

$$\alpha(e, \varphi, f, a)(\sigma) := \begin{cases} f(\sigma, u, \bar{e})(\mathsf{id}_1) & \text{if } u : \varphi(\sigma) \text{ is found} \\ a(\sigma) & \text{otherwise} \end{cases}$$

for $\sigma : \mathbf{C}(1, c)$. Then by definition $\forall_{c':\mathbf{C}} \forall_{\sigma:\mathbf{C}(c',c)} \forall_{u:\varphi(\sigma)} f(\sigma, u, \bar{e}) = \alpha(e, \varphi, f, a)\sigma$. ◀

▶ **Proposition 18.** *Suppose that $\mathbf{C}(1, \mathbb{I})$ only contains 0 and 1, namely $\forall_{\sigma:\mathbf{C}(1,\mathbb{I})} \sigma = 0 \vee \sigma = 1$. Then for every type $\Gamma \vdash A$ in $\mathcal{S}$, there exists a term*

$$\Delta\Gamma \vdash p : \prod_{a_0,a_1:\nabla A} \mathsf{Path}(\nabla A, a_0, a_1)$$

in $\mathbf{PSh}(\mathbf{C})$.

Proof. We may assume that Γ is the empty context. The term p corresponds to a natural transformation that takes an object $c : \mathbf{C}$, elements $a_0, a_1 : \nabla A(c)$ and a morphism $i : \mathbf{C}(c, \mathbb{I})$ and returns an element $p(a_0, a_1, i) : \nabla A(c)$ such that $p(a_0, a_1, 0) = a_0$ and $p(a_0, a_1, 1) = a_1$. We define $p(a_0, a_1, i) : \mathbf{C}(1, c) \to A$ as

$$p(a_0, a_1, i)(\sigma) := \begin{cases} a_0(\sigma) & \text{if } i\sigma = 0 \\ a_1(\sigma) & \text{if } i\sigma = 1 \end{cases}$$

for $\sigma : \mathbf{C}(1, c)$. Then by definition $p(a_0, a_1, 0) = a_0$ and $p(a_0, a_1, 1) = a_1$. ◀

5 A Failure of Propositional Resizing in Cubical Assemblies

An *assembly*, also called an *ω-set*, is a set A equipped with a non-empty set $E_A(a)$ of natural numbers for every $a \in A$. When $n \in E_A(a)$, we say n is a *realizer* for a or n *realizes* a. A morphism $f : A \to B$ of assemblies is a function $f : A \to B$ between the underlying sets such that there exists a partial recursive function e such that, for any $a \in A$ and $n \in E_A(a)$, the application en is defined and belongs to $E_B(f(a))$. In that case we say f is *tracked* by e or e is a *tracker* of f. We shall denote by **Asm** the category of assemblies and morphisms of assemblies. Note that assemblies can be defined in terms of partial combinatory algebras instead of natural numbers and partial recursive functions [44], and that the rest of this section works for assemblies on any non-trivial partial combinatory algebra.

The category **Asm** is a model of dependent type theory. Contexts are interpreted as assemblies. Types $\Gamma \vdash A$ are interpreted as families of assemblies $(A(\gamma) \in \mathbf{Asm})_{\gamma \in \Gamma}$ indexed over the underlying set of Γ. Terms $\Gamma \vdash a : A$ are interpreted as sections $a \in \prod_{\gamma \in \Gamma} A(\gamma)$ such that there exists a partial recursive function e such that, for any $\gamma \in \Gamma$ and $n \in E_\Gamma(\gamma)$, the application en is defined and belongs to $E_{A(\gamma)}(a(\gamma))$. For a type $\Gamma \vdash A$, the context extension $\Gamma.A$ is interpreted as an assembly $(\sum_{\gamma \in \Gamma} A(\gamma), (\gamma, a) \mapsto \{\langle n, m\rangle \mid n \in E_\Gamma(\gamma), m \in E_{A(\gamma)}(a)\})$ where $\langle n, m\rangle$ is a fixed effective encoding of tuples of natural numbers. It is known that **Asm** supports dependent product types, dependent sum types, extensional identity types, unit type, disjoint finite coproducts and natural numbers. See, for example, [44, 28, 25]. For a family of assemblies A over Γ, the propositional truncation $\|A\|$ is the family

$$\|A\|(\gamma) = \begin{cases} \{*\} & \text{if } A(\gamma) \neq \emptyset \\ \emptyset & \text{if } A(\gamma) = \emptyset \end{cases}$$

with realizers $E_{\|A\|(\gamma)}(*) = \bigcup_{a \in A(\gamma)} E_{A(\gamma)}(a)$.

It is also well-known that **Asm** has an impredicative universe **PER**. It is an assembly whose underlying set is the set of partial equivalence relations, namely symmetric and transitive relations, on $\mathbb{N}$ and the set of realizers of R is $E_{\mathbf{PER}}(R) = \{0\}$. The type $\mathbf{PER} \vdash \mathsf{el}_{\mathbf{PER}}$ is defined as $\mathsf{el}_{\mathbf{PER}}(R) = \mathbb{N}/R$, the set of R-equivalence classes on $\{n \in \mathbb{N} \mid R(n, n)\}$ with realizers $E_{\mathbb{N}/R}(\xi) = \xi$. The universe **PER** classifies *modest families*. An assembly A is said to be *modest* if $E_A(a)$ and $E_A(a')$ are disjoint for distinct $a, a' \in A$. By definition $\mathbb{N}/R$ is modest for every $R \in \mathbf{PER}$. Conversely, for a modest assembly A, one can define a partial equivalence relation R such that $A \cong \mathbb{N}/R$. For the impredicativity of **PER**, see [23, 28, 25].

The category **Asm** satisfies the hypotheses of Theorem 14 with impredicative universe **PER**, propositional universe **2** and the internal category $\mathbf{B}_{\mathsf{ord}}$ defined in Section 4.4. We will refer to the presheaf model of cubical type theory generated by these structures as the *cubical assembly model*.

5.1 Propositional Resizing

In cubical type theory, a type $\Gamma \vdash A$ is a *homotopy proposition* if the type $\Gamma, a_0, a_1 : A \vdash \mathsf{Path}(A, a_0, a_1)$ has an inhabitant. For a universe $\mathcal{U}$, we define the universe of homotopy propositions as

$$\mathsf{hProp}_{\mathcal{U}} := \sum_{A:\mathcal{U}} \prod_{a_0, a_1 : A} \mathsf{Path}(A, a_0, a_1).$$

Following the HoTT book [33], we regard $\mathsf{hProp}_{\mathcal{U}}$ as a subtype of $\mathcal{U}$.

The *propositional resizing axiom* [33, Section 3.5] asserts that, for nested universes $\mathcal{U} : \mathcal{U}'$, the inclusion $\mathsf{hProp}_{\mathcal{U}} \to \mathsf{hProp}_{\mathcal{U}'}$ is an equivalence. When $\mathcal{U}$ is an impredicative universe, we define

$$A : \mathsf{hProp}_{\mathcal{U}'} \vdash A^* := \prod_{X : \mathsf{hProp}_{\mathcal{U}}} (A \to X) \to X : \mathsf{hProp}_{\mathcal{U}}$$

$$A : \mathsf{hProp}_{\mathcal{U}'} \vdash \eta_A := \lambda a.\lambda X f.fa : A \to A^*.$$

If η_A is an equivalence for any $A : \mathsf{hProp}_{\mathcal{U}'}$, then the inclusion $\mathsf{hProp}_{\mathcal{U}} \to \mathsf{hProp}_{\mathcal{U}'}$ is an equivalence by univalence. Conversely, if the inclusion $\mathsf{hProp}_{\mathcal{U}} \to \mathsf{hProp}_{\mathcal{U}'}$ is an equivalence, then one can find $A' : \mathsf{hProp}_{\mathcal{U}}$ and $e : A \simeq A'$ from $A : \mathsf{hProp}_{\mathcal{U}'}$. Then we have a function $\lambda\alpha.e^{-1}(\alpha A' e) : A^* \to A$, and thus η_A is an equivalence because both A and A^* are homotopy propositions. Note that the construction $A \mapsto (A^*, \eta_A)$ works for any homotopy proposition A and is independent of the choice of the upper universe $\mathcal{U}'$. Therefore, we can formulate the propositional resizing axiom in cubical type theory with an impredicative universe as follows.

▶ **Axiom 19.** *For every homotopy proposition $\Gamma \vdash A$, the function $\Gamma \vdash \eta_A : A \to A^*$ is an equivalence.*

We will show that the cubical assembly model does not satisfy Axiom 19.

▶ Remark 20. We focus on resizing propositions into the impredicative universe. The cubical assembly model also has predicative universes, assuming the existence of Grothendieck universes in the metatheory. It remains an open question whether the predicative universes in the cubical assembly model satisfy the propositional resizing axiom.

5.2 Uniform Objects

The key idea to a counterexample to propositional resizing is the orthogonality of modest and *uniform* assemblies [44]: if X is modest and A is uniform and well-supported, then the map $\lambda xa.x : X \to (A \to X)$ is an isomorphism. Since the impredicative universe **PER** classifies modest assemblies, $\prod_{X:\mathbf{PER}}(A \to X) \to X$ is always inhabited for a uniform, well-supported assembly A. We extend the notion of uniformity for internal presheaves in **Asm**.

An assembly A is said to be *uniform* if $\bigcap_{a \in A} E_A(a)$ is non-empty. We say an internal presheaf A on an internal category $\mathbf{C}$ is *uniform* if every $A(c)$ is uniform. An internal presheaf A on $\mathbf{C}$ is said to be *well-supported* if the unique morphism into the terminal presheaf is regular epi. For an internal presheaf A, the following are equivalent:

- A is well-supported;
- $\|A\|$ is the terminal presheaf;
- there exists a partial recursive function e such that, for any $c \in \mathbf{C}_0$ and $n \in E_{\mathbf{C}_0}(c)$, there exists an $a \in A(c)$ such that en is defined and belongs to $E_A(a)$.

By definition a modest assembly cannot distinguish elements with a common realizer, while elements of a uniform assembly have a common realizer. Thus a modest assembly "believes a uniform assembly has at most one element". Formally, the following proposition holds.

▶ **Proposition 21.** *Let $\mathbf{C}$ be a category in* **Asm**. *For a uniform internal presheaf A on $\mathbf{C}$ and an internal functor $X : \mathbf{C}^{\mathrm{op}} \to \mathbf{PER}$, the precomposition function*

$$i^* : (\|A\| \to X) \to (A \to X)$$

is an isomorphism, where $i : A \to \|A\|$ is the constructor for propositional truncation. In particular, if, in addition, A is well-supported, then the function $\lambda xa.x : X \to (A \to X)$ is an isomorphism.

Proof. Since i is regular epi, i^* is a monomorphism. Hence it suffices to show that i^* is regular epi. Let k_c denote a common realizer of $A(c)$, namely $k_c \in \bigcap_{a \in A(c)} E(a)$. Let $c \in \mathbf{C}_0$ be an object and $x : \mathbf{y}c \times A \to X$ a morphism of presheaves tracked by e. We have to show that there exists a morphism $\hat{x} : \mathbf{y}c \times \|A\| \to X$ such that $\hat{x} \circ (\mathbf{y}c \times i) = x$ and that a tracker of $\hat{x}$ is computable from the code of e. For any $\sigma : c' \to c$ and $a, a' \in A(c')$, we have $enk_{c'} \in E(x(\sigma, a)) \cap E(x(\sigma, a'))$ for some $n \in E(\sigma)$. Since $X(c')$ is modest, we have $x(\sigma, a) = x(\sigma, a')$. Hence x induces a morphism of presheaves $\hat{x} : \mathbf{y}c \times \|A\| \to X$ tracked by e such that $\hat{x} \circ (\mathbf{y}c \times i) = x$. ◀

▶ **Theorem 22.** *Let $\Gamma \vdash A$ be a type in the cubical assembly model. Suppose that A is uniform and well-supported as an internal presheaf on $\mathbf{El}(\Gamma)$ and does not have a section. Then the function $\Gamma \vdash \eta : A \to A^*$ is not an equivalence.*

Proof. By Proposition 21, we see that $A^* = \prod_{X : \mathsf{hProp}} (A \to X) \to X$ has an inhabitant while A does not have an inhabitant by assumption. ◀

▶ **Theorem 23.** *Let $\Gamma \vdash A$ be a type in $\mathbf{Asm}$. Suppose that A is uniform and well-supported but does not have a section. Then the function $\Delta\Gamma \vdash \eta : \nabla A \to (\nabla A)^*$ is not an equivalence.*

Proof. By Theorem 22, it suffices to show that the type $\Delta\Gamma \vdash \nabla A$ is uniform and well-supported but does not have a section. For the uniformity, let k_γ be a common realizer of $A(\gamma)$ for $\gamma \in \Gamma$. For any object $c \in \mathbf{C}$ and element $\gamma \in \Gamma$, the code of the constant function $n \mapsto k_\gamma$ is a common realizer of $\nabla A(c, \gamma) = \mathbf{C}(1, c) \to A(\gamma)$.

For the well-supportedness, let e be a partial recursive function such that, for any γ and $n \in E_\Gamma(\gamma)$, there exists an $a \in A(\gamma)$ such that en is defined and belongs to $E_{A(\gamma)}(a)$. Then the function f mapping (n, x) to the code of the function $y \mapsto ex$ realizes that ∇A is well-supported. Indeed, for any $c \in \mathbf{C}$, $n \in E_{\mathbf{C}}(c)$, $\gamma \in \Gamma$ and $x \in E_\Gamma(\gamma)$, the code $f(n, x)$ realizes the constant function $\mathbf{C}(1, c) \ni \sigma \mapsto a \in A(\gamma)$ for some $a \in A(\gamma)$ such that $ex \in E_{A(\gamma)}(a)$.

Finally ∇A does not have a section because $\nabla A(1) \cong A$ and A does not have a section. ◀

5.3 The Counterexample

We define an assembly Γ to be $(\mathbb{N}, n \mapsto \{m \in \mathbb{N} \mid m > n\})$ and a family of assemblies A on Γ as $A(n) = (\{m \in \mathbb{N} \mid m > n\}, m \mapsto \{n, m\})$. Then A is uniform because every $A(n)$ has a common realizer n. The identity function realizes that A is well-supported. To see that A does not have a section, suppose that a section $f \in \prod_{n \in \Gamma} A(n)$ is tracked by a partial recursive function e. Then for any $m > n$, we have $em \in \{n, f(n)\}$. This implies that $m \leq e(m+1) \leq f(0)$ for any m, a contradiction. Note that this construction of $\Gamma \vdash A$ works for any non-trivial partial combinatory algebra C because natural numbers can be effectively encoded in C.

Since $\mathbf{B}_{\mathsf{ord}}(1, \mathbb{I}) \cong \mathbf{2}$ only contains end-points, the type $\Delta\Gamma \vdash \nabla A$ in the cubical assembly model is a fibration and homotopy proposition by Propositions 17 and 18, while by Theorem 23 the function $\Delta\Gamma \vdash \eta : \nabla A \to (\nabla A)^*$ is not an equivalence. Hence the propositional resizing axiom fails in the cubical assembly model.

6 Conclusion and Future Work

We have formulated the axioms for modeling cubical type theory in an elementary topos given by Orton and Pitts [30] in a weaker setting and explained how to construct a model of

cubical type theory in a category satisfying those axioms. As a striking example, we have constructed a model of cubical type theory with an impredicative and univalent universe in the category of cubical assemblies which is not an elementary topos. It has turned out that this impredicative universe in the cubical assembly model does not satisfy the propositional resizing axiom.

There is a natural question: can we construct a model of type theory with a univalent and impredicative universe satisfying the propositional resizing axiom? One possible approach to this question is to consider a full subcategory of the category of cubical assemblies in which every homotopy proposition is equivalent to some modest family. Benno van den Berg [43] constructed a model of a variant of homotopy type theory with a univalent and impredicative universe of 0-types that satisfies the propositional resizing axiom. Roughly speaking he uses a category of degenerate trigroupoids in the category of *partitioned assemblies* [44], and thus the category of cubical partitioned assemblies is a candidate for such a full subcategory. However, the model given in [43] only supports weaker forms of identity types and dependent product types, and it is unclear whether it can be seen as a model of ordinary homotopy type theory.

Higher inductive types are another important feature of homotopy type theory. One can construct some higher inductive types including propositional truncation in the cubical assembly model [42], internalizing the construction of higher inductive types in cubical sets [12] using W-types with reductions [41]. An open question, raised by Steve Awodey, is whether these higher inductive types are equivalent to their impredicative encodings.

The cubical assembly model is a realizability-based model of type theory with higher dimensional structures, but it does not seem to be what should be called a *realizability* ∞*-topos*, a higher dimensional analogue of a realizability topos [44]. One problem is that, in the cubical assembly model, realizers seem to play no role in its internal cubical type theory, because the existence of a realizer of a homotopy proposition does not imply the existence of a section of it. Indeed, the cubical assembly model does not satisfy Church's Thesis [42] which holds in the effective topos [22]. One can nevertheless find a left exact localization of the cubical assembly model in which Church's Thesis holds [42].

Our construction of models of cubical type theory is a syntactic one following Orton and Pitts [30]. The original idea of using the internal language of a topos to construct models of cubical type theory was proposed by Coquand [11]. There are also semantic and categorical approaches. Frumin and van den Berg [16] presented a way of constructing a model structure on a full subcategory of an elementary topos with a path connection algebra, which is essentially same as the model structure on the category of fibrant cubical sets described by Spitters [37]. Since they make no essential use of subobject classifiers, we conjecture that one can construct a model structure on a full subcategory of a suitable locally cartesian closed category with a path connection algebra. Sattler [34], based on his earlier work with Gambino [17], gave a construction of a right proper combinatorial model structure on a suitable category with an interval object. Although Gambino and Sattler use Garner's small object argument [18] which requires the cocompleteness of underlying categories, their construction is expected to work for non-cocomplete categories such as the category of cubical assemblies using Swan's small object argument over codomain fibrations [40, 41].

References

1 Steve Awodey. Impredicative encodings in HoTT, 2017. Talk at the workshop "Computer-aided mathematical proof". URL: `http://www.newton.ac.uk/seminar/20170711090010001`.

2 Steve Awodey, Jonas Frey, and Sam Speight. Impredicative Encodings of (Higher) Inductive Types. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 76–85, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209130.

3 Steven Awodey and Andrej Bauer. Propositions As [Types]. *J. Log. and Comput.*, 14(4):447–471, August 2004. doi:10.1093/logcom/14.4.447.

4 Andrej Bauer, Jason Gross, Peter LeFanu Lumsdaine, Michael Shulman, Matthieu Sozeau, and Bas Spitters. The HoTT Library: A Formalization of Homotopy Type Theory in Coq. In *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs*, CPP 2017, pages 164–172, New York, NY, USA, 2017. ACM. doi:10.1145/3018610.3018615.

5 Marc Bezem, Thierry Coquand, and Simon Huber. A Model of Type Theory in Cubical Sets. In Ralph Matthes and Aleksy Schubert, editors, *19th International Conference on Types for Proofs and Programs (TYPES 2013)*, volume 26 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 107–128, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2013.107.

6 Marc Bezem, Thierry Coquand, and Simon Huber. The Univalence Axiom in Cubical Sets. *Journal of Automated Reasoning*, 63(2):159–171, August 2019. doi:10.1007/s10817-018-9472-6.

7 Lars Birkedal, Ranald Clouston, Bassel Mannaa, Rasmus Ejlers Møgelberg, Andrew M. Pitts, and Bas Spitters. Modal Dependent Type Theory and Dependent Right Adjoints, 2019. arXiv:1804.05236v3.

8 Ulrik Buchholtz and Edward Morehouse. Varieties of Cubical Sets. In Peter Höfner, Damien Pous, and Georg Struth, editors, *Relational and Algebraic Methods in Computer Science: 16th International Conference, RAMiCS 2017, Lyon, France, May 15-18, 2017, Proceedings*, pages 77–92. Springer International Publishing, Cham, 2017. doi:10.1007/978-3-319-57418-9_5.

9 J.W. Cartmell. *Generalised algebraic theories and contextual categories*. PhD thesis, Oxford University, 1978.

10 Cyril Cohen, Thierry Coquand, Simon Huber, and Anders Mörtberg. Cubical Type Theory: A Constructive Interpretation of the Univalence Axiom. In Tarmo Uustalu, editor, *21st International Conference on Types for Proofs and Programs (TYPES 2015)*, volume 69 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 5:1–5:34, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.TYPES.2015.5.

11 Thierry Coquand. Internal version of the uniform Kan filling condition, 2015. URL: http://www.cse.chalmers.se/~coquand/shape.pdf.

12 Thierry Coquand, Simon Huber, and Anders Mörtberg. On Higher Inductive Types in Cubical Type Theory. In *Proceedings of the 33rd Annual ACM/IEEE Symposium on Logic in Computer Science*, LICS '18, pages 255–264, New York, NY, USA, 2018. ACM. doi:10.1145/3209108.3209197.

13 Thierry Coquand and Gérard Huet. The Calculus of Constructions. *Information and Computation*, 76(2):95–120, 1988. doi:10.1016/0890-5401(88)90005-3.

14 Peter Dybjer. Internal Type Theory. In Stefano Berardi and Mario Coppo, editors, *Types for Proofs and Programs: International Workshop, TYPES '95 Torino, Italy, June 5–8, 1995 Selected Papers*, pages 120–134. Springer Berlin Heidelberg, Berlin, Heidelberg, 1996. doi:10.1007/3-540-61780-9_66.

15 Jonas Frey. Towards a realizability model of homotopy type theory, 2017. Talk at CT 2017. URL: http://www.mat.uc.pt/~ct2017/slides/frey_j.pdf.

16 Dan Frumin and Benno van den Berg. A homotopy-theoretic model of function extensionality in the effective topos. *Mathematical Structures in Computer Science*, pages 1–27, 2018. doi:10.1017/S0960129518000142.

17 Nicola Gambino and Christian Sattler. The Frobenius condition, right properness, and uniform fibrations . *Journal of Pure and Applied Algebra*, 221(12):3027–3068, 2017. doi:10.1016/j.jpaa.2017.02.013.

18 Richard Garner. Understanding the Small Object Argument. *Applied Categorical Structures*, 17(3):247–285, June 2009. doi:10.1007/s10485-008-9137-4.

19 Jean-Yves Girard, Yves Lafont, and Paul Taylor. *Proofs and Types*, volume 7 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 1989. URL: http://www.paultaylor.eu/stable/Proofs+Types.html.

20 Martin Hofmann and Thomas Streicher. Lifting Grothendieck Universes, 1997. URL: http://www.mathematik.tu-darmstadt.de/~streicher/NOTES/lift.pdf.

21 Martin Hofmann and Thomas Streicher. The groupoid interpretation of type theory. In *Twenty-five years of constructive type theory (Venice, 1995)*, volume 36 of *Oxford Logic Guides*, pages 83–111. Oxford Univ. Press, New York, 1998.

22 J.M.E. Hyland. The Effective Topos. In A.S. Troelstra and D. van Dalen, editors, *The L. E. J. Brouwer Centenary Symposium*, volume 110 of *Studies in Logic and the Foundations of Mathematics*, pages 165–216. Elsevier, 1982. doi:10.1016/S0049-237X(09)70129-6.

23 J.M.E. Hyland. A small complete category. *Annals of Pure and Applied Logic*, 40(2):135–165, 1988. doi:10.1016/0168-0072(88)90018-8.

24 Bart Jacobs. Comprehension categories and the semantics of type dependency. *Theoretical Computer Science*, 107(2):169–207, 1993. doi:10.1016/0304-3975(93)90169-T.

25 Bart Jacobs. *Categorical Logic and Type Theory*. Elsevier Science, 1st edition, 1999.

26 Chris Kapulkin and Peter LeFanu Lumsdaine. The Simplicial Model of Univalent Foundations (after Voevodsky), 2018. arXiv:1211.2851v5.

27 Daniel R. Licata, Ian Orton, Andrew M. Pitts, and Bas Spitters. Internal Universes in Models of Homotopy Type Theory. In Hélène Kirchner, editor, *3rd International Conference on Formal Structures for Computation and Deduction (FSCD 2018)*, volume 108 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 22:1–22:17, Dagstuhl, Germany, 2018. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.FSCD.2018.22.

28 Giuseppe Longo and Eugenio Moggi. Constructive natural deduction and its "ω-set" interpretation. *Mathematical Structures in Computer Science*, 1(2):215–254, 1991. doi:10.1017/S0960129500001298.

29 Per Martin-Löf. An Intuitionistic Theory of Types: Predicative Part. *Studies in Logic and the Foundations of Mathematics*, 80:73–118, 1975. doi:10.1016/S0049-237X(08)71945-1.

30 Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. In Jean-Marc Talbot and Laurent Regnier, editors, *25th EACSL Annual Conference on Computer Science Logic (CSL 2016)*, volume 62 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:19, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. doi:10.4230/LIPIcs.CSL.2016.24.

31 Ian Orton and Andrew M. Pitts. Axioms for Modelling Cubical Type Theory in a Topos. *Logical Methods in Computer Science*, 14, December 2018. doi:10.23638/LMCS-14(4:23)2018.

32 Wesley Phoa. An introduction to fibrations, topos theory, the effective topos and modest sets. Technical Report ECS-LFCS-92-208, The University of Edinburgh, 2006. URL: http://www.lfcs.inf.ed.ac.uk/reports/92/ECS-LFCS-92-208/.

33 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. http://homotopytypetheory.org/book/, Institute for Advanced Study, 2013.

34 Christian Sattler. The Equivalence Extension Property and Model Structures, 2017. arXiv:1704.06911v4.

35 Michael Shulman. Higher Inductive Types via Impredicative Polymorphism, 2011. URL: https://homotopytypetheory.org/2011/04/25/higher-inductive-types-via-impredicative-polymorphism/.

36 Sam Speight. Impredicative Encodings of Inductive Types in Homotopy Type Theory. Master's thesis, Carnegie Mellon University, 2017. URL: http://www.cs.ox.ac.uk/people/sam.speight/publications/sams-hott-thesis.pdf.

37 Bas Spitters. Cubical sets and the topological topos, 2016. arXiv:1610.05270v1.

38 Wouter Pieter Stekelenburg. Constructive Simplicial Homotopy, 2016. arXiv:1604.04746v1.

39 Andrew W. Swan. An Algebraic Weak Factorisation System on 01-Substitution Sets: A Constructive Proof. *Journal of Logic & Analysis*, 8:1–35, 2016. doi:10.4115/jla.2016.8.1.

40 Andrew W. Swan. Lifting Problems in Grothendieck Fibrations, 2018. arXiv:1802.06718v1.

41 Andrew W. Swan. W-Types with Reductions and the Small Object Argument, 2018. arXiv:1802.07588v1.

42 Andrew W. Swan and Taichi Uemura. On Church's Thesis in Cubical Assemblies, 2019. arXiv:1905.03014v1.

43 Benno van den Berg. Univalent polymorphism, 2018. arXiv:1803.10113v2.

44 Jaap van Oosten. *Realizability: An Introduction to Its Categorical Side*, volume 152 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, San Diego, USA, 2008.

45 Vladimir Voevodsky. A universe polymorphic type system, 2012. URL: https://ncatlab.org/ufias2012/files/Universe+polymorphic+type+sytem.pdf.

A Details of Composition for Gluing and Universe

We give explicit definitions of composition operations for gluing and universes described in Section 3.4.

Before that, we introduce some notations. for a fibration $\Gamma, i : \mathbb{I} \vdash A(i)$, one can derive the *composition operation*

$$\frac{\Gamma \vdash \varphi : \mathsf{Cof} \qquad \Gamma, i : \mathbb{I} \vdash f(i) : \varphi \to A(i) \qquad \Gamma \vdash a : A(e) \qquad \Gamma, u : \varphi \vdash f(e)u = a}{\Gamma \vdash \mathsf{comp}^i_e(A(i), f(i), a) : A(\bar{e})}$$

such that $\Gamma, u : \varphi \vdash f(\bar{e})u = \mathsf{comp}^i_e(A(i), f(i), a)$. Concretely, for a fibration structure $\alpha : \mathsf{Fib}(A)$, we define

$$\gamma : \Gamma \vdash \mathsf{comp}^i_e(A(i), f(i), a) := \alpha(\lambda i.(\gamma, i), e, \varphi, \lambda u i. f(i)u, a).$$

In the notation $\mathsf{comp}^i_e(A(i), f(i), a)$, the variable i is considered to be bound. Usually we use the composition operation in the form of

$$\mathsf{comp}^i_e(A(i), [(u_1 : \varphi_1) \mapsto g_1(u_1, i), \dots, (u_n : \varphi_n) \mapsto g_n(u_n, i)], a)$$

with a system $[(u_1 : \varphi_1) \mapsto g_1(u_1, i), \dots, (u_n : \varphi_n) \mapsto g_n(u_n, i)] : \varphi_1 \vee \dots \vee \varphi_n \to A(i)$.

A.1 Some Derived Notions and Operations

We recall some notions and operations derivable in cubical type theory without gluing and universes.

Composition operations are preserved by function application [10, Section 5.2]: one can derive an operation

$$\frac{\Gamma, i : \mathbb{I} \vdash h(i) : A(i) \to B(i) \qquad \Gamma \vdash e : \mathbf{2} \qquad \Gamma \vdash \varphi : \mathsf{Cof} \qquad \Gamma, i : \mathbb{I} \vdash f(i) : \varphi \to A(i) \qquad \Gamma \vdash a : A(e) \qquad \Gamma, u : \varphi \vdash f(e)u = a}{\Gamma \vdash \mathsf{pres}^i_e(h(i), f(i), a) : \mathsf{Path}(B(\bar{e}), c_1, c_2)}$$

such that $\Gamma, u : \varphi, j : \mathbb{I} \vdash h(\bar{e})(f(\bar{e})u) = \mathsf{pres}^i_e(h(i), f(i), a)j$, where $c_1 = \mathsf{comp}^i_e(B(i), h(i) \circ f(i), h(e)a)$ and $c_2 = h(\bar{e})(\mathsf{comp}^i_e(A(i), f(i), a))$.

Equivalences are characterized by a kind of extension property [10, Section 5.3]: for fibrations $\Gamma \vdash A$ and $\Gamma \vdash B$, one can derive an operation

$$\frac{\begin{array}{c}\Gamma \vdash f : A \simeq B \\ \Gamma \vdash e : \mathbf{2} \qquad \Gamma \vdash \varphi : \mathsf{Cof} \qquad \Gamma \vdash b : B \qquad \Gamma \vdash p : \varphi \to \sum_{a:A} \mathsf{Path}(B, b, fa)\end{array}}{\Gamma \vdash \mathsf{equiv}(f, p, b) : \sum_{a:A} \mathsf{Path}(B, b, fa)}$$

such that $\Gamma, u : \varphi \vdash pu = \mathsf{equiv}(f, p, b)$.

For a fibration $\Gamma, i : \mathbb{I} \vdash A(i)$, we define a function called *transport* $\Gamma, e : \mathbf{2} \vdash \mathsf{tp}^i_e(A(i)) : A(e) \to A(\bar{e})$ to be $\mathsf{tp}^i_e(A(i))a = \mathsf{comp}^i_e(A(i), [], a)$. This function $\mathsf{tp}^i_e(A(i))$ is an equivalence [10, Section 7.1].

A.2 Gluing

Proof of Proposition 8. Let $p : \mathbb{I} \to \Gamma$, $e : \mathbf{2}$, $\psi : \mathsf{Cof}$, $g : \psi \to \prod_{i:\mathbb{I}} \prod_{u:\varphi(pi)} A(u)$, $h : \psi \to \prod_{i:\mathbb{I}} B(pi)$, $a : \prod_{u:\varphi(pe)} A(u)$ and $b : B(pe)$, and suppose $\forall_{v:\psi} \forall_{i:\mathbb{I}} \forall_{u:\varphi(pi)} f(u)(gviu) = hvi$, $\forall_{u:\varphi(pe)} f(u)(au) = b$ and $\forall_{v:\psi} gve = a \wedge hve = b$. We have to find elements $\bar{a} : \prod_{u:\varphi(p\bar{e})} A(u)$ and $\bar{b} : B(p\bar{e})$ such that $\forall_{u:\varphi(p\bar{e})} f(u)(\bar{a}u) = \bar{b}$ and $\forall_{v:\psi} gv\bar{e} = \bar{a} \wedge hv\bar{e} = \bar{b}$. We define

$$\begin{aligned}
\bar{b}_1 &:= \mathsf{comp}^i_e(B(pi), [(v : \psi) \mapsto hvi], b) : B(p\bar{e}) \\
\delta &:= \forall_{i:\mathbb{I}} \varphi(pi) : \mathsf{Cof} \\
\bar{a}_1 &:= \lambda w. \mathsf{comp}^i_e(A(wi), [(v : \psi) \mapsto gvi(wi)], a(we)) : \prod_{w:\delta} A(w\bar{e}) \\
q &: \prod_{w:\delta} \mathsf{Path}(\bar{b}_1, f(w\bar{e})(\bar{a}_1 w)) \\
qw &:= \mathsf{pres}^i_e(f(wi), [(v : \psi) \mapsto gvi(wi)], a(we)) \\
\bar{a} &: \prod_{u:\varphi(p\bar{e})} A(u) \\
q_2 &: \prod_{u:\varphi(p\bar{e})} \mathsf{Path}(\bar{b}_1, f(u)(\bar{a}u)) \\
(\bar{a}u, q_2 u) &:= \mathsf{equiv}(f(u), [(w : \delta) \mapsto (\bar{a}_1 w, qw), (v : \psi) \mapsto (gv\bar{e}u, \lambda i.\bar{b}_1)], \bar{b}_1) \\
\bar{b} &:= \mathsf{comp}^i_0(B(p\bar{e}), [(u : \varphi(p\bar{e})) \mapsto q_2 ui, (v : \psi) \mapsto hv\bar{e}], \bar{b}_1) : B(p\bar{e})
\end{aligned}$$

Then one can derive that $\bar{b} = q_2 u 1 = f(u)(\bar{a}u)$ for $u : \varphi(p\bar{e})$ and that $\bar{a} = gv\bar{e}$ and $\bar{b} = hv\bar{e}$ for $v : \psi$. Moreover, for every $w : \prod_{i:\mathbb{I}} \varphi(pi)$, we have $\bar{a}(w\bar{e}) = \bar{a}_1 w = \mathsf{comp}^i_e(A(wi), [(v : \psi) \mapsto gvi(wi)], a(we))$ which means the preservation of fibration structure by the function $\Gamma, u : \varphi \vdash \lambda(a, b).au : \mathsf{Glue}(\varphi, f) \to A(u)$. ◀

A.3 Universes

Proof of Proposition 9. Let $e : \mathbf{2}$, $\varphi : \mathsf{Cof}$, $f : \varphi \to \mathbb{I} \to \mathcal{U}^F$ and $B : \mathcal{U}^F$ such that $\forall_{u:\varphi} fue = B$. We have to find a $\bar{B} : \mathcal{U}^F$ such that $\forall_{u:\varphi} fu\bar{e} = \bar{B}$. Let $A := \lambda u. fu\bar{e} : \varphi \to \mathcal{U}^F$. We have an equivalence $g := \lambda u. \mathsf{tp}^i_{\bar{e}}(fui) : \prod_{u:\varphi} Au \simeq B$. Let $\bar{B} := \mathsf{SGlue}(\varphi, g) : \mathcal{U}^F$, then $\forall_{u:\varphi} fu\bar{e} = Au = \bar{B}$. ◀