20th International Conference on Types for Proofs and Programs

TYPES'14, May 12–15, 2014, Paris, France

Edited by
Hugo Herbelin
Pierre Letouzey
Matthieu Sozeau

LIPIcs – Vol. 39 – TYPES 2014 www.dagstuhl.de/lipics

Editors

Hugo Herbelin
πr^2 team, P.P.S. laboratory
INRIA & Université Paris Diderot
France
hugo.herbelin@inria.fr

Pierre Letouzey
πr^2 team, P.P.S. laboratory
INRIA & Université Paris Diderot
France
pierre.letouzey@inria.fr

Matthieu Sozeau
πr^2 team, P.P.S. laboratory
INRIA & Université Paris Diderot
France
matthieu.sozeau@inria.fr

ACM Classification 1998
D.1.1 Applicative (Functional) Programming, D.2.4 Software/Program Verification, F.3.1 Specifying and Verifying and Reasoning about Programs, F.4.1 Mathematical Logic

ISBN 978-3-939897-88-0

Published online and open access by
Schloss Dagstuhl – Leibniz-Zentrum für Informatik GmbH, Dagstuhl Publishing, Saarbrücken/Wadern, Germany. Online available at http://www.dagstuhl.de/dagpub/978-3-939897-88-0.

Publication date
October, 2015

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

Digital Object Identifier: 10.4230/LIPIcs.TYPES.2014.i

ISBN 978-3-939897-88-0 **ISSN 1868-8969** **http://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

http://www.dagstuhl.de/lipics

Contents

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau

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

Preface

The 20th International Conference on Types for Proofs and Programs (TYPES'14) was held in Paris, France from May 12 to May 16, 2014, consisting of the main conference and one satellite event[1].

The conference was attended by about a hundred scientists. The responsible persons for local organisation were Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau, and the program committee for the selection of the conference presentations consisted of:

- Andreas Abel, Chalmers University of Technology and Gothenburg University, Sweden
- Andrej Bauer, Fakulteta za matematiko in fiziko, Ljubljana, Slovenia
- Małgorzata Biernacka, University of Wroclaw, Poland
- Lars Birkedal, Aarhus University, Denmark
- Paul Blain Levy, University of Birmingham, UK
- Herman Geuvers, Radboud University and Eindhoven University of Technology, Netherlands
- Hugo Herbelin, INRIA Paris-Rocquencourt, France (co-chair)
- Pierre Letouzey, University Paris-Diderot, France (co-chair)
- Ralph Matthes, IRIT, CNRS and University of Toulouse, France
- Conor McBride, University of Strathclyde, UK
- Luís Pinto, University of Minho, Braga, Portugal
- Claudio Sacerdoti, University of Bologna, Italy
- Aleksy Schubert, University of Warsaw, Poland
- Matthieu Sozeau, INRIA Paris-Rocquencourt, France (co-chair)
- Thomas Streicher, TU Darmstadt, Germany

The TYPES meetings were first organised in the late 1980's and were supported by a series of EU programmes from 1989 to 2008. Previous meetings were held in Antibes (1990), Edinburgh (1991), Båstad (1992), Nijmegen (1993), Båstad (1994), Turin (1995), Aussois (1996), Kloster Irsee (1998), Lökeberg (1999), Durham (2000), Berg en Dal (2002), Turin (2003), Jouy-en-Josas (2004), Nottingham (2006), Cividale del Friuli (2007), Turin (2008), Aussois (2009), Warsaw (2010), Bergen (2011) and Toulouse (2013).

Three invited talks and 39 contributed talks were given at the meeting, and we got 17 submissions to these open post-proceedings, out of which 14 papers were accepted. All papers obtained at least two reviews, and up to six reviews, counting a second round of review.

As editors of this post-proceedings volume, we would like to thank the authors of the paper submissions, whether accepted or not. And we gratefully acknowledge all 31 anonymous external referees for their valuable work. The overall very high quality of the resulting papers is in part due to their careful reading and commenting on the obtained material.

The TYPES'14 conference received financial and/or logistic support from the Institut Henri Poincaré (IHP), INRIA Paris-Rocquencourt, and Université Paris Diderot. The EasyChair platform was used for reviewing and it was a very agreeable experience as editors of these post-proceedings to work with Marc Herbstritt and Schloss Dagstuhl.

Hugo Herbelin, Pierre Letouzey and Matthieu Sozeau
July 2015

[1] PCC: Proof, Computation, Complexity.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Authors Index

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

Terminal Semantics for Codata Types in Intensional Martin-Löf Type Theory*

Benedikt Ahrens and Régis Spadotti

Institut de Recherche en Informatique de Toulouse
Université Paul Sabatier, Toulouse, France

Abstract

We study the notions of *relative comonad* and *comodule over a relative comonad*. We use these notions to give categorical semantics for the coinductive type families of streams and of infinite triangular matrices and their respective cosubstitution operations in intensional Martin-Löf type theory. Our results are mechanized in the proof assistant `Coq`.

1998 ACM Subject Classification F.3.2 Semantics of Programming Languages

Keywords and phrases relative comonad, Martin-Löf type theory, coinductive type, computer theorem proving

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.1

1 Introduction

In this work, we study the notions of *relative comonad* and *comodule over a relative comonad*. We then use these notions for a case study in categorical semantics of coinductive data types in intensional Martin-Löf type theory (IMLTT): we characterize two coinductive type families and their respective cosubstitution operations in IMLTT via a universal property. The first codata type we consider is the *homogeneous* type family of streams, parametrized by a base type. The second one is the *heterogeneous* codata type family of infinite triangular matrices parametrized by the type of diagonal entries. In the rest of the introduction, we explain some of the vocabulary occurring in these first sentences: in Section 1.1 we briefly introduce (intensional) Martin-Löf type theory. In Section 1.2 we discuss inductive types and their semantics, before passing to *co*inductive types in Section 1.3. We describe the difference between homogeneous and heterogeneous (co)data types in Section 1.4 and explain substitution for leaf-labeled trees and cosubstitution for node-labeled trees in Section 1.5. Sections 1.6 to 1.8 of the introduction concern more "administrative" aspects.

1.1 Intensional Martin-Löf type theory

Martin-Löf type theory (MLTT) [17] is a dependent type theory developed in the 1970's, which provides a foundation of mathematics. It is based on the Curry-Howard isomorphism, that is, it treats logic as a fragment of the general type theory. There are two notions of "sameness" in Martin-Löf type theory, an external (judgmental) one, and an internal (propositional) one. The latter is given by the *Martin-Löf identity type*, which is an inductively defined binary relation on any given type. Its only constructor relates any term to itself, that is, the relation defined by the Martin-Löf identity type is the least reflexive relation on

* The work of Benedikt Ahrens was partially supported by the CIMI (Centre International de Mathématiques et d'Informatique) Excellence program ANR-11-LABX-0040-CIMI within the program ANR-11-IDEX-0002-02 during a postdoctoral fellowship.

a given type. Judgmentally equal terms are always propositionally equal, but the converse is not always true. Indeed, one distinguishes two variants of type theory: in *extensional* type theory, propositional equalities (i.e. terms of identity type) are reflected, via a *reflection rule*, into the judgmental equality of the type theory. Here, judgmental and propositional equality thus coincide. *Intensional* Martin-Löf type theory lacks the reflection principle for the sake of decidability of type checking. This variant forms the basis of the computer proof assistants `Coq`, `Matita` and `Agda`.

1.2 Wellfounded trees and their semantics

A tree is a special kind of graph (with labeled edges and unlabeled vertices), consisting of a *root* (edge), to which a number of *subtrees* are attached. The subtrees are themselves trees. The root of a tree and the roots of any subtrees are more generally called *nodes*. We gather trees of a fixed shape into a set or type, depending on the foundations we work in. The possible shapes of trees are given by two pieces of data:

1. the set/type of potential nodes and
2. the number of subtrees attached to each node.

Such a pair is called a *signature*, and we are interested in the set/type of trees which are formed according to a given signature.

A tree is called *wellfounded* if all of its subtrees have finite length. In this section we consider the set/type of wellfounded trees of a given signature. Such a set/type is called *inductively generated* by the signature. Intuitively, this set/type is the least one stable under forming trees starting from leaves and attaching already constructed trees as subtrees to nodes. One might also consider *potentially infinite*, i.e., non-wellfounded trees; these are called *co*inductive and are the main object of this work. We discuss them in Section 1.3.

Many objects in mathematics and logic can be represented as (wellfounded) *trees*. For instance, a natural number can be considered as such a tree: let $\{S, Z\}$ be the set of possible nodes, such that node S always has one subtree and node Z does not have any subtrees, i.e. Z is a *leaf*. The natural number 2 is represented by the tree $S \longrightarrow S \longrightarrow Z$. This example demonstrates the fundamental aspect of those tree-like objects: they can be used to model basic mathematical and logical objects.

By "semantics of inductive types" we mean a mathematical description of the set/type of trees of a given signature. More precisely, the goal is to give a category-theoretic explanation to the above description as "the least set/type stable under forming trees". However, the question of how to give such a characterization depends on the formal system we work in.

In a *set-theoretic* setting, inductive sets are characterized as initial algebras for some endofunctor on the category of sets [16].

In a *type-theoretic* setting as given by Martin-Löf type theory [17], two approaches to the semantics of inductive types have been studied: one approach consists in showing that inductive types exist in a *model* of the type theory (see, e.g., [19]). Another approach is to prove that adding certain type-theoretic rules to the type theory – rules which postulate the existence of inductive types in the type theory – implies (or is equivalent to) the existence of a universal object *within* type theory (see, e.g., [12, 9]). This latter approach is the one we adopt in the present work: we add axioms to intensional Martin-Löf type theory, postulating the existence of *co*inductive types.

The reflection rule equips *extensional* type theory with extensional features similar to those of set theory. As a consequence, the characterization given by Dybjer [12] of an inductive type in extensional MLTT works as in set theory: an inductive type constitutes the initial algebra for some endofunctor on the category of types.

In *intensional* Martin-Löf type theory the characterization of inductive types as initial algebras of some endofunctor fails for a lack of function extensionality [12], but it can be recovered [9] in an *extension* of intensional MLTT called *Homotopy Type Theory* (HoTT) [23]. In this extension, function extensionality is provable from the *Univalence Axiom*. For a suitable definition of *uniqueness* – *contractibility* in HoTT jargon – one can prove a logical equivalence between type-theoretic rules specifying an inductive type on the one hand and the existence of an initial algebra within the theory on the other hand. The mentioned work [9] thus shows that the characterization of inductive types as initial algebras carries over from extensional to intensional type theory if one adds an extensionality principle for functions and adapts the notion of uniqueness.

1.3 Non-wellfounded trees: simply dualizing?

Coinductive sets/types describe sets/types of potentially *infinite* trees, that is, trees that may have an infinite chain of subtrees – still formed according to some signature. An example of such a coinductive set/type is given by the *co*natural numbers, which are specified by the same signature as the natural numbers. They consist of the elements of the natural numbers and an additional, infinite, tree $S \longrightarrow S \longrightarrow S \longrightarrow \dots$.

In a traditional set-theoretic setting, the theory of coinductive sets is completely dual to that of inductive sets: *co*inductive sets are characterized as terminal coalgebras of suitable endofunctors [16].

In intensional Martin-Löf type theory, this duality between inductive and coinductive objects breaks. This is rooted in the unsuitability of the Martin-Löf identity type to express sameness for inhabitants of coinductive types: while identity terms are inductively generated and hence necessarily finite, a proof of sameness between coinductive terms – which represent infinite objects – constitutes a potentially infinite object itself. The type of such proofs hence cannot be exhaustively given by the (inductive) identity type.

Instead of comparing two coinductive terms in IMLTT modulo identity, one defines a binary coinductive relation called *bisimilarity* on a given coinductive type, with respect to which one compares its inhabitants [11]. Expressed categorically, bisimilarity in IMLTT is given as a *weakly terminal* relation on the coinductive type that is compatible with the coalgebra structure. Consequently, we consider two maps into a coinductive type to be the same if they are *pointwise bisimilar* – an analogue to the aforementioned principle of function extensionality available in HoTT.

With these conventions, we give, in the present work, a characterization of some coinductive data types as *terminal* object in some category defined in IMLTT. More precisely, we consider an example of *homogeneous* codata type, streams, and an example of *heterogeneous* codata type, triangular matrices. For each of these examples we prove, from type-theoretic rules specifying the respective codata type added to the basic rules of IMLTT, the existence of a terminal object in some category *within IMLTT*. The objects of the considered categories are not plain coalgebras for some endofunctor, but rather coalgebras with extra structure. This extra structure accounts for a *cosubstitution* operation with which the considered codata types are endowed in a canonical way. The cosubstitution operation is explained in Section 1.5.

In the present work, we do not study *existence* of coinductive types. In variants of IMLTT, coinductive types have been derived from inductive types: for IMLTT with Uniqueness of Identity Proofs [7], and for IMLTT augmented by the Univalence Axiom [4].

1.4 Homogeneous and heterogeneous trees

Instead of considering types of trees of a given signature, we consider, more generally, *families* $T(X)$ of types of trees, families parametrized by a type X. We call a tree $t : T(X)$ over X *homogeneous* if all of its subtrees are also trees over X, i.e. if all the subtrees are elements of $T(X)$. A signature accordingly is called homogeneous if it only admits homogeneous trees. On the other hand, a tree t is called *heterogeneous*, if its subtrees t_i ($i : I$ for some indexing type I) are trees over type $F_i(X)$ for maps of types $F_i : \mathsf{Type} \to \mathsf{Type}$. Obviously, when the functors F_i are identity functors, we get back homogeneous trees; heterogeneous trees are thus more general than homogeneous trees.

In the present work, we study examples of both homogeneous and heterogeneous trees: a *stream* is a *homogeneous* tree with one subtree over the same type of nodes, whereas an *infinite triangular matrix* is a *heterogeneous* tree (over a type, say, A) with one subtree over type $E \times A$ for a fixed, but arbitrary type E. This is explained in more detail below.

The examples we consider are trees with just one subtree. A generalization to trees (no matter whether homogeneous or heterogeneous) with a family of subtrees indexed by some fixed type I as above is straightforward – we refrain from striving for this additional generality in order to keep the exposition as simple and clear as possible, and also in view of a lack of practical examples where this generality would be useful.

1.5 Cosubstitution: a comonadic operation on parametrized codata

We first consider the case of *leaf-labeled trees*. Let $T(X)$ be a type family of trees (of a given shape) with *leaves from* X, i.e., $t : T(X)$ is a tree (of a given shape) with leaves in X. Let furthermore $f : X \to T(Y)$ be a map. We obtain a tree $t' : T(Y)$ by replacing any leaf of t by its image under the map f. This replacement operation is called *(simultaneous) substitution* – "simultaneous" because all the leaves are substituted at once. A well-studied example is that of the lambda calculus, the terms of which can be formalized as leaf-labeled trees over a type of *free variables*, that is, over a *context* [8]. More precisely, let $\mathsf{LC}(X)$ be the type of lambda terms in context X. A map $f : X \to \mathsf{LC}(Y)$ is called a *substitution rule*, indicating how to substitute any free variable $x : X$ occurring in a term $t : \mathsf{LC}(X)$. The substitution operation (parametrized by contexts X and Y) takes as input a lambda term $t : \mathsf{LC}(X)$ and a rule $f : X \to \mathsf{LC}(Y)$ and returns the substituted term $t' : \mathsf{LC}(Y)$. The categorical characterization of this substitution operation has been studied extensively; our list of pointers [8, 13, 20, 22, 14, 6, 3] is necessarily incomplete. One such characterization is via *monads*; substitution can be seen as part of a monadic structure on the functor given by the parametrized data type [8, 14, 6, 3].

Node-labeled trees can be equipped with a *cosubstitution* operation. Is this cosubstitution operation part of a comonad structure? In a set-theoretic setting, the fact that cosubstitution for coinductive sets is comonadic is established by Uustalu and Vene [24].

If we consider infinite (i.e., coinductive) node-labeled trees in IMLTT, however, the algebraic laws of cosubstitution for such trees have to be considered *modulo bisimilarity rather than identity*. For this we develop the notion of *relative comonad* and *comodule over a relative comonad*. Indeed, we show that our exemplary codata type families of streams and triangular matrices constitute relative comonads, and the maps which return the substream resp. submatrix of a given stream resp. matrix constitute comodule morphisms over those relative comonads.

Our goal is to characterize those codata types and their respective cosubstitution operation as a universal object in some category. Traditionally, codata types are characterized as

terminal coalgebras of some endofunctor. However, this characterization does not account for the cosubstitution operation. In order to integrate cosubstitution into the categorical semantics of the coinductive types under consideration, we thus define a more complicated category of "models" of the signature specifying those types – coalgebras with extra structure accounting for a comonadic operation. The terminal such model is given by the codata type together with its cosubstitution operation. Our terminal semantics thus characterizes not only the codata types themselves but also the bisimilarity relation and – via the use of (relative) comonads – a canonical cosubstitution operation on them.

1.6 Computer-checked proofs

All our results have been implemented in the proof assistant Coq [10]. The Coq source files and HTML documentation are available online [5]. Here, we hence omit the proofs and focus on definitions and statements.

1.7 Type theory vs. set theory

The category-theoretic concepts studied in this work are agnostic to the foundational system being worked in. While we present them in a type-theoretic style, the definitions and lemmas can trivially be transferred to a set-theoretic setting. Throughout this article, we use type-theoretic notation, writing $t : T$ to indicate that t is of type T. For instance, we write $f : \mathcal{C}(A, B)$ to indicate that f is a morphism from object A to object B in category $\mathcal{C}$. Whenever an operation takes several arguments, we write some of them as indices; these indices might be omitted when they can be deduced from the type of the later arguments. We assume basic knowledge of category theory; any instances used are defined in the following.

1.8 Organisation of the paper

In Section 2 we introduce some concepts and notations used later on. In Section 3 we present the coinductive type families Stream of streams and Tri of infinite triangular matrices and some operations on those codata types, in particular, the respective cosubstitution operations. In Section 4 we present *relative comonads* and define the category of comonads relative to a fixed functor. We give some examples of relative comonads using the codata types presented in Section 3, and relate relative comonads to traditional comonads. In Section 5 we define *comodules over relative comonads* and give some constructions of comodules. Again, examples of comodules are taken from Section 3. In Section 6 we define categories of models for the (signatures of the) codata types presented in Section 3, based on the category-theoretic notions developed in the preceding sections. We then prove that the codata types constitute the terminal models in the respective categories. We present an example of a map defined as a terminal morphism in the category of models for streams, exploiting terminality of Stream. In Section 7 we give an overview of the formalization of this work in the proof assistant Coq.

2 Preliminaries

We present a few particular category-theoretic objects used later on, and fix some notation.

▶ **Definition 1** (Setoids in IMLTT)**.** A **setoid** in intensional Martin-Löf type theory is a pair $(A, \sim_A)$ of a type A together with an equivalence relation $\sim_A$ on A. Given a setoid $S = (A, \sim_A)$, we often abuse notation by writing $s : S$ instead of $s : A$ for a term s of A. Given two setoids $(A, \sim_A)$ and $(B, \sim_B)$, the cartesian product $A \times B$ of their underlying

types is equipped with an equivalence relation given component-wise, thus yielding a product on setoids.

▶ **Definition 2** (Category in IMLTT, [2, 15])**.** A **category** $\mathcal{C}$ in intensional Martin-Löf type theory is given by

- a type of objects, also denoted by $\mathcal{C}$;
- for any two objects $a, b : \mathcal{C}$, a *setoid* $\mathcal{C}(a, b)$ of morphisms from a to b;
- an identity morphism $\mathsf{id}_a : \mathcal{C}(a, a)$ for any $a : \mathcal{C}$;
- a dependent composition operation $(\circ)_{a,b,c} : \mathcal{C}(b, c) \times \mathcal{C}(a, b) \to \mathcal{C}(a, c)$;
- a proof that composition is compatible with the equivalence relations of the setoids of morphisms;
- proofs of unitality and associativity of composition stated in terms of the equivalence relations on the morphisms.

We write $f : a \to b$ for $f : \mathcal{C}(a, b)$, when the category $\mathcal{C}$ can be deduced from the context.

Functors and natural transformations are defined in terms of the setoidal equivalence relations on morphisms. We omit the definitions, which can be found in our `Coq` files [5].

▶ **Definition 3** (Some categories in IMLTT)**.** We denote by Type the category (in the sense of Definition 2) of types (of a fixed universe) and total functions between them in Martin-Löf type theory. The setoidal equivalence relation on $\mathsf{Type}(A, B)$ is given by pointwise equality as given by the Martin-Löf identity type, $f \sim g \ =_{\mathrm{def}} \ \forall x : A, fx = gx$.

We denote by Setoid the category (in the sense of Definition 2) an object of which is a setoid. A morphism between setoids is a type-theoretic function between the underlying types that is compatible in the obvious sense with the equivalence relations of the source and target setoids. The equivalence relation on setoid morphisms is given by pointwise equivalence: two parallel morphisms of setoids $f, g : A \to B$ are equivalent if for any $a : A$ we have $fa \sim_B ga$. The category Setoid thus is cartesian closed.

▶ **Definition 4** (A functor from types to setoids)**.** The functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ is defined as the left adjoint to the forgetful functor $U : \mathsf{Setoid} \to \mathsf{Type}$. Explicitly, the functor eq sends any type X to the setoid $(X, =_X)$ given by the type X itself, equipped with the propositional equality relation $=_X$ specified via Martin-Löf's identity type on X.

▶ **Remark 5** (Notation for product)**.** *We denote the category-theoretic binary product of objects A and B of a category $\mathcal{C}$ by $A \times B$. We write $\mathsf{pr}_1(A, B) : \mathcal{C}(A \times B, A)$ and $\mathsf{pr}_2(A, B) : \mathcal{C}(A \times B, B)$ for the projections, occasionally omitting the argument (A, B). Given $f : \mathcal{C}(A, B)$ and $g : \mathcal{C}(A, C)$, we write $\langle f, g\rangle : \mathcal{C}(A, B \times C)$ for the induced map into the product such that $\mathsf{pr}_1 \circ \langle f, g\rangle = f$ and $\mathsf{pr}_2 \circ \langle f, g\rangle = g$.*

Both of the categories of Definition 3 have finite products, i.e., binary products and a terminal object.

▶ **Definition 6** (Product-preserving functor)**.** A functor $F : \mathcal{C} \to \mathcal{D}$ between categories with finite products **preserves finite products** if it preserves the terminal object and, for any two objects A and B of $\mathcal{C}$, the morphism $\phi^F_{A,B}$ is an isomorphism, where

$$\phi^F_{A,B} := \big\langle F(\mathsf{pr}_1), F(\mathsf{pr}_2)\big\rangle : F(A \times B) \to FA \times FB \ .$$

▶ **Example 7.** The functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ of Definition 4 preserves finite products.

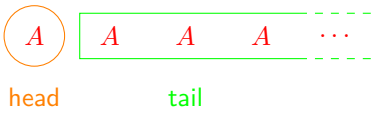


Figure 1 A stream decomposes into its head and tail.

3 Codata types in intensional Martin-Löf type theory

In this section, two coinductive type families in intensional Martin-Löf type theory (IMLTT) [17] are presented. For $a, b : A$, the identity type between a and b is denoted by $a = b$.

We present the type-theoretic rules specifying these codata types and *bisimilarity* on them. Categorically, bisimilarity on a given codata type is given by the largest binary relation on that type that is compatible (in a sense to be specified later) with the *observations* on that codata; it is the appropriate notion of sameness on inhabitants of these types [11]. Here, an observation is the result of *destructing*, i.e. taking apart, an element of the codata type. For instance, an infinite list a.k.a. a stream (defined in detail in Example 8) over a base type A allows to observe the first element – the head – of that list (which is a term of type A) and, corecursively, the tail of that list, which is again an infinite list over A.

A coinductive type together with its associated bisimilarity relation forms a setoid as in Definition 3. We thus denote bisimilar elements t and t' by $t \sim t'$.

A map into a codata type is defined using its *introduction* rule. Intuitively, the introduction rule takes as arguments the *observations* one can make on the image of the map thus defined. For instance, a map into streams is defined by specifying head and tail on the output of that map: this intuition can be obtained by considering the combination of introduction and computation rules from Figure 2. We thus use a kind of "copattern matching" [1] to define maps into streams (and analogously for other codata types): the definition $f := \mathsf{corec}\ h\ t$ is written as the pair of clauses

$$\mathsf{head} \circ f := h \quad \text{and} \quad \mathsf{tail} \circ f := t \ .$$

The first example is the type of *streams* of elements of a given base type A:

▶ **Example 8** (The coinductive type family of streams)**.** Let A be a type. A *stream over* A is, intuitively, an infinite list of elements of A. Given such a stream t, we can extract an element of A, called $\mathsf{head}(t)$ and the rest of the stream, called $\mathsf{tail}(t)$, which is again a stream over A; see Figure 1 for a schematic illustration of the two operations.

Formally, the type – setoid, actually – $\mathsf{Stream}A$ of *streams over* A is coinductively defined via the destructors head and tail given in Figure 2. A map $f : T \to \mathsf{Stream}A$ from an arbitrary type T into the type of streams is specified by giving, for each possible input $t : T$, an element $a : A$ – the head of the image stream – and another $t' : T$ – an element which is mapped to tail $f(t)$. This intuition is made formal by the introduction and computation rules of Figure 2.

The second part of Figure 2 introduces *bisimilarity* as a binary relation on streams (over a fixed type A) that is compatible with the observations in the sense that bisimilar streams have equal heads and bisimilar tails (cf. the elimination rules). We call a *bisimulation on streams (over a given type)* any binary relation that is compatible with the observations on streams. If two streams are related by *any* bisimulation, then they are also bisimilar; this is what the introduction rule declares. In that sense, bisimilarity is the largest bisimulation.

Rules for Stream

Formation

$$\frac{A : \mathsf{Type}}{\mathsf{Stream}A : \mathsf{Type}}$$

Elimination

$$\frac{t : \mathsf{Stream}A}{\mathsf{head}_A\ t : A} \qquad \frac{t : \mathsf{Stream}A}{\mathsf{tail}_A\ t : \mathsf{Stream}A}$$

Introduction

$$\frac{T : \mathsf{Type} \qquad hd : T \to A \qquad tl : T \to T}{\mathsf{corec}_A\ hd\ tl : T \to \mathsf{Stream}A}$$

Computation

$$\frac{hd : T \to A \qquad tl : T \to T \qquad t : T}{\mathsf{head}_A(\mathsf{corec}_A\ hd\ tl\ t) = hd(t)}$$

$$\frac{hd : T \to A \qquad tl : T \to T \qquad t : T}{\mathsf{tail}_A(\mathsf{corec}_A\ hd\ tl\ t) = \mathsf{corec}_A\ hd\ tl\ (tl\ t)}$$

Bisimilarity on Stream

Formation

$$\frac{A : \mathsf{Type} \qquad s, t : \mathsf{Stream}A}{\mathsf{bisim}_A\ s\ t : \mathsf{Type}}$$

Elimination

$$\frac{s, t : \mathsf{Stream}A \qquad p : \mathsf{bisim}_A\ s\ t}{\mathsf{head}_A\ s = \mathsf{head}_A\ t} \qquad \frac{s, t : \mathsf{Stream}A \qquad p : \mathsf{bisim}_A\ s\ t}{\mathsf{bisim}_A(\mathsf{tail}_A\ s)(\mathsf{tail}_A\ t)}$$

Introduction

$$\frac{\begin{array}{r}\vdash R : \mathsf{Stream}A \to \mathsf{Stream}A \to \mathsf{Type} \\ x, y : \mathsf{Stream}A \vdash R\ x\ y \to \mathsf{head}\ x = \mathsf{head}\ y \\ x, y : \mathsf{Stream}A \vdash R\ x\ y \to R\ (\mathsf{tail}\ x)(\mathsf{tail}\ y)\end{array}}{x, y : \mathsf{Stream}A \vdash R\ x\ y \to \mathsf{bisim}\ x\ y}$$

Figure 2 Rules for streams and bisimilarity on them.

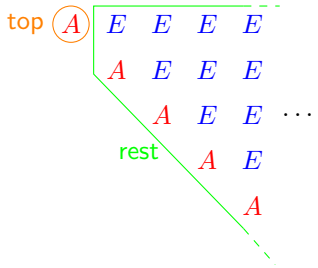


Figure 3 An infinite triangular matrix over type A and its destructors top and rest.

We define a *co*substition operation on streams via the following clauses:

$$\begin{aligned}\mathsf{cosubst}_{A,B} &: (\mathsf{Stream}A \to B) \to \mathsf{Stream}A \to \mathsf{Stream}B\\ \mathsf{head} \circ \mathsf{cosubst}\ f &:= f \quad \text{and}\\ \mathsf{tail} \circ \mathsf{cosubst}\ f &:= \mathsf{cosubst}\ f \circ \mathsf{tail}\ .\end{aligned}$$

According to our convention above, by this, we actually mean

$$\mathsf{cosubst}_{A,B}(f) := \mathsf{corec}\ f\ \mathsf{tail}\ .$$

We call this operation "cosubstitution" since its type is dual to the simultaneous substitution operation of the lambda calculus [8]. Cosubstitution is compatible with bisimilarity on streams, and thus is (the carrier of) a family

$$\mathsf{cosubst}_{A,B} : \mathsf{Setoid}(\mathsf{Stream}A, \mathsf{eq}B) \to \mathsf{Setoid}(\mathsf{Stream}A, \mathsf{Stream}B)\ .$$

Streams are node-labeled trees where every node has exactly one subtree. We also consider a type of trees where every node has an arbitrary, but fixed, number of subtrees, parametrized by a type B.

▶ **Example 9** (Node-labeled trees)**.** We denote by $\mathsf{Tree}_B(A)$ the codata type given by one destructor head and a family of destructors $(\mathsf{tail}_b)_{b:B}$ with type analogous to that defining tail of Example 8. We thus obtain Stream by considering, for B, the singleton type.

We also consider the *heterogeneous* codata type family of *infinite triangular matrices*:

▶ **Example 10.** Infinite triangular matrices are studied in detail by Matthes and Picard [18]. We give a brief summary, but urge the reader to consult the given reference for an in-depth explanation. The codata type family Tri of infinite triangular matrices is parametrized by a fixed type E for entries not on the diagonal, and indexed by another, *variable*, type A for entries on the diagonal. Schematically, such a matrix looks like in Figure 3.

The codata type is specified via two destructors top and rest, whose types are given in Figure 4.

Given a matrix over type A, its rest – obtained by removing the first element on the diagonal, i.e. the top element – can be considered as a trapezium as indicated by the green line in Figure 3, or alternatively, as a triangular matrix over type $E \times A$, by bundling the entries of the diagonal with those above as indicated by the orange frames, shown in Figure 5. The latter representation is reflected in the type of the destructor rest.

This change of parameter (from A to $E \times A$) in the type of rest is the reason why the family Tri is called *heterogeneous*.

Rules for Tri

Formation

$$\frac{A : \mathsf{Type}}{\mathsf{Tri}A : \mathsf{Type}}$$

Elimination

$$\frac{t : \mathsf{Tri}A}{\mathsf{top}_A\ t : A} \qquad \frac{t : \mathsf{Tri}A}{\mathsf{rest}_A\ t : \mathsf{Tri}(E \times A)}$$

Introduction

$$\frac{T : \mathsf{Type} \to \mathsf{Type} \qquad hd : \forall A, TA \to A \qquad tl : \forall A, TA \to T(E \times A)}{\mathsf{corec}_T\ hd\ tl : \forall A, TA \to \mathsf{Tri}A}$$

Computation

$$\frac{hd : \forall A, TA \to A \qquad tl : \forall A, TA \to T(E \times A) \qquad t : TA}{\mathsf{top}_T(\mathsf{corec}_A\ hd\ tl\ t) = hd(t)}$$

$$\frac{hd : \forall A, TA \to A \qquad tl : \forall A, TA \to T(E \times A) \qquad t : TA}{\mathsf{rest}_T(\mathsf{corec}_A\ hd\ tl\ t) = \mathsf{corec}_A\ hd\ tl\ (tl\ t)}$$

Bisimilarity for Tri

Formation

$$\frac{A : \mathsf{Type} \qquad s, t : \mathsf{Tri}A}{\mathsf{bisim}_A\ s\ t : \mathsf{Type}}$$

Elimination

$$\frac{s, t : \mathsf{Tri}A \qquad p : \mathsf{bisim}_A\ s\ t}{\mathsf{top}_A\ s = \mathsf{top}_A\ t} \qquad \frac{s, t : \mathsf{Tri}A \qquad p : \mathsf{bisim}_A\ s\ t}{\mathsf{bisim}_A(\mathsf{rest}_A\ s)(\mathsf{rest}_A\ t)}$$

Introduction

$$\frac{\begin{array}{r} A : \mathsf{Type} \vdash R : \mathsf{Tri}A \to \mathsf{Tri}A \to \mathsf{Type} \\ A : \mathsf{Type}, x, y : \mathsf{Tri}A \vdash R\ x\ y \to \mathsf{top}\ x = \mathsf{top}\ y \\ A : \mathsf{Type}, x, y : \mathsf{Tri}A \vdash R\ x\ y \to R\ (\mathsf{tail}\ x)(\mathsf{tail}\ y) \end{array}}{A : \mathsf{Type}, x, y : \mathsf{Tri}A \vdash R\ x\ y \to \mathsf{bisim}\ x\ y}$$

Figure 4 Rules for triangular matrices and bisimilarity on them.

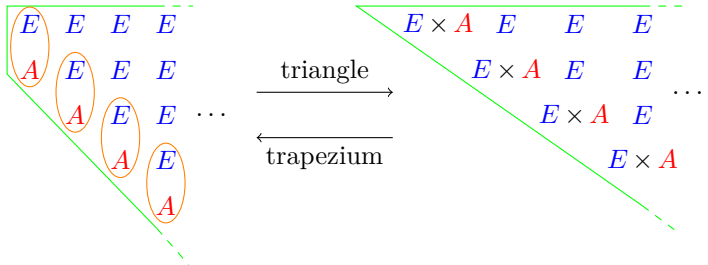


Figure 5 The rest of a matrix over A is a matrix over $E \times A$ (illustration taken from [18]).

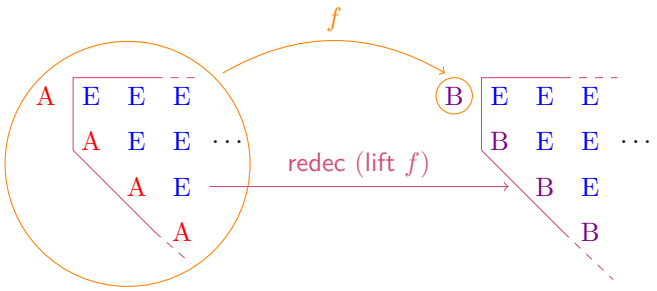


Figure 6 Definition of redecoration (illustration taken from [18]).

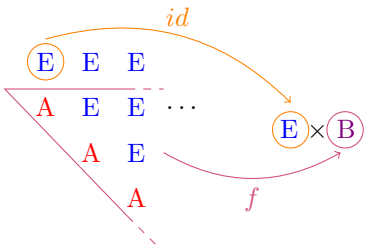


Figure 7 Definition of lifting (illustration taken from [18]).

A cosubstitution operation, "redecoration", $\mathsf{redec}_{A,B} : (\mathsf{Tri}A \to B) \to \mathsf{Tri}A \to \mathsf{Tri}B$ is defined through the clauses

$$\mathsf{top} \circ \mathsf{redec}\ f := f \quad \text{and} \quad \mathsf{rest} \circ \mathsf{redec}\ f := \mathsf{redec}\ (\mathsf{lift}\ f) \circ \mathsf{rest}\ . \tag{1}$$

An illustration of the redecoration operation is given in Figure 6 (taken from [18]).

Here, the need for the family of auxiliary functions

$$\mathsf{lift}_{A,B} : (\mathsf{Tri}A \to B) \to \mathsf{Tri}(E \times A) \to E \times B$$

arises from the *heterogeneity* of the destructor rest. These functions are suitably defined to account for the change of the type of the argument of redec when redecorating $\mathsf{rest}\ t : \mathsf{Tri}(E \times A)$ rather than $t : \mathsf{Tri}A$, namely

$$\mathsf{lift}(f) := \langle \mathsf{pr}_1(E, A) \circ \mathsf{top}_{E \times A}, f \circ \mathsf{cut}_A \rangle\ .$$

An illustration of this operation is given in Figure 7 (taken from [18]).

The auxiliary function $\mathsf{cut}_A : \mathsf{Tri}(E \times A) \to \mathsf{Tri}A$ cuts the upper row of elements in E of a trapezium (equivalently, of a matrix over $E \times A$). It is defined corecursively via

$$\mathsf{top} \circ \mathsf{cut} := \mathsf{pr}_2 \circ \mathsf{top} \quad \text{and} \quad \mathsf{rest} \circ \mathsf{cut} := \mathsf{cut} \circ \mathsf{rest}\ .$$

All the operations are suitably compatible with the bisimilarity relations, so that they can be equipped with the types

$$\begin{aligned}\mathsf{redec}_{A,B} &: \mathsf{Setoid}(\mathsf{Tri}A, \mathsf{eq}B) \to \mathsf{Setoid}(\mathsf{Tri}A, \mathsf{Tri}B)\\ \mathsf{lift}_{A,B} &: \mathsf{Setoid}(\mathsf{Tri}A, \mathsf{eq}B) \to \mathsf{Setoid}\big(\mathsf{Tri}(E \times A), \mathsf{eq}(E \times B)\big)\\ \mathsf{cut}_A &: \mathsf{Setoid}(\mathsf{Tri}(E \times A), \mathsf{Tri}A)\ .\end{aligned}$$

▶ **Remark 11.** *We do not specify any computation rules for bisimilarity, i.e., we think of bisimilarity as a proof-irrelevant notion. The lack of such a computation rule amounts to asserting that bisimilarity is a* weakly *terminal bisimulation on the respective codata.*

4 Relative comonads and their morphisms

In this section we define *comonads relative to a functor* and morphisms between those, and present some examples.

Relative monads were defined by Altenkirch, Chapman, and Uustalu [6] as a notion of monad-like structure whose underlying functor is not necessarily an endofunctor. An example of relative monad given in that work is the lambda calculus over finite contexts.

The dual notion is that of a relative *co*monad, more precisely, a comonad relative to some functor $F : \mathcal{C} \to \mathcal{D}$. Indeed, since the functor underlying a relative comonad is not necessarily endo, one requires an auxiliary functor, which "mediates" between source and target category. An application of this auxiliary functor is inserted where necessary to make the comonad-like operations and axioms welltyped.

▶ **Definition 12.** Let $F : \mathcal{C} \to \mathcal{D}$ be a functor. A **relative comonad** T **over** F is given by

- a map $T : \mathcal{C}_0 \to \mathcal{D}_0$ on the objects of the categories involved and
- operations

$$\begin{aligned}&\mathsf{counit} : \forall A : \mathcal{C}_0, \mathcal{D}(TA, FA)\\ &\mathsf{cobind} : \forall A, B : \mathcal{C}_0, \mathcal{D}(TA, FB) \to \mathcal{D}(TA, TB)\end{aligned}$$

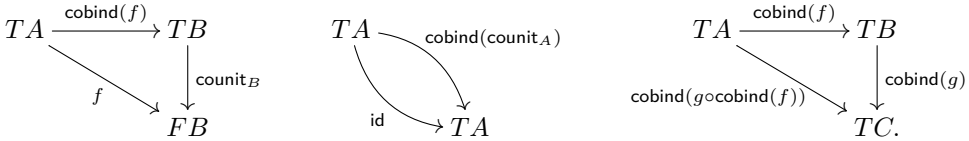
such that the following equations hold:

$$\forall A, B : \mathcal{C}_0, \forall f : \mathcal{D}(TA, FB), \mathsf{counit}_B \circ \mathsf{cobind}(f) = f \tag{2}$$

$$\forall A : \mathcal{C}_0, \mathsf{cobind}(\mathsf{counit}_A) = \mathsf{id}_{TA} \tag{3}$$

$$\forall A, B, C : \mathcal{C}_0, \forall f : \mathcal{D}(TA, FB), \forall g : \mathcal{D}(TB, FC),\\ \mathsf{cobind}(g) \circ \mathsf{cobind}(f) = \mathsf{cobind}(g \circ \mathsf{cobind}(f))\ , \tag{4}$$

in diagrammatic form:



Definition 12 does not mention an action of T on morphisms. Indeed, just like with relative monads, this action is definable rather than part of the definition:

▶ **Definition 13.** Let T be a comonad relative to $F : \mathcal{C} \to \mathcal{D}$. For $f : \mathcal{C}(A, B)$ we define

$$T(f) := \mathsf{cobind}(Ff \circ \mathsf{counit}_A) : \mathcal{D}(TA, TB)\ .$$

The functor properties are easily checked.

▶ **Remark 14.** *It follows from the comonadic axioms that counit and cobind are natural transformations with respect to the functoriality defined in Definition 13:*

$$\mathsf{counit} : T \dot{\to} F : \mathcal{C} \to \mathcal{D}$$
$$\mathsf{cobind} : \mathcal{D}(T\_, F\_) \dot{\to} \mathcal{D}(T\_, T\_) : \mathcal{C}^{op} \times \mathcal{C} \to \mathsf{Setoid} \ .$$

Comonads relative to the identity functor are exactly (traditional) comonads.

We obtain a comonad relative to $F : \mathcal{C} \to \mathcal{D}$ from a comonad on the category $\mathcal{C}$ by composing M with F:

▶ **Example 15** (Relative comonads from comonads)**.** Let $F : \mathcal{C} \to \mathcal{D}$ be a fully faithful functor and $(M, \mathsf{counit}, \mathsf{cobind})$ be a (traditional) comonad (in Kleisli form) on $\mathcal{C}$. We define a comonad FM relative to F by setting:

$$\begin{aligned} FM(A) &:= F(MA) \\ \mathsf{counit}^{FM}_A &:= F(\mathsf{counit}^M_A) : \mathcal{D}(FMA, FA) \\ \mathsf{cobind}^{FM}_{A,B}(f) &:= F\big(\mathsf{cobind}^M_{A,B}(F^{-1}f)\big) \ . \end{aligned}$$

The other way round does not require any conditions on the functor F:

▶ **Example 16** (Relative comonads from comonads II)**.** Let $F : \mathcal{C} \to \mathcal{D}$ be a functor and $(M, \mathsf{counit}, \mathsf{cobind})$ be a (traditional) comonad (in Kleisli form) on $\mathcal{D}$. We define a comonad MF relative to F by setting:

$$\begin{aligned} MF(A) &:= M(FA) \\ \mathsf{counit}^{MF}_A &:= \mathsf{counit}^M_{FA} : \mathcal{D}(MFA, FA) \\ \mathsf{cobind}^{MF}_{A,B}(f) &:= \mathsf{cobind}^M_{FA,FB}(f) : \mathcal{D}(MFA, MFB) \quad \text{for} \quad f : \mathcal{D}(MFA, FB) \ . \end{aligned}$$

The next examples concern the codata type families presented in Section 3.

▶ **Example 17** (Streams)**.** The codata type family $\mathsf{Stream} : \mathsf{Type} \to \mathsf{Setoid}$ of Example 8 is equipped with a structure of a comonad relative to the functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ with $\mathsf{counit}_A := \mathsf{head}_A$ and $\mathsf{cobind}_{A,B} := \mathsf{cosubst}_{A,B}$.

▶ **Remark 18.** *Instead of considering* Stream *as a relative comonad from* Type *to* Setoid*, it could be considered as a traditional comonad on the category* Setoid*, propagating the equivalence relation on the base type, say A, to* Stream$A$*. The relative point of view can then be recovered as an instance of Example 16.*

▶ **Example 19** (Trees)**.** Fix a type B. Analogously to Example 17, the map $A \mapsto \mathsf{Tree}_B(A)$ of Example 9 is equipped with a structure of a comonad relative to $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$.

▶ **Example 20** (Infinite triangular matrices)**.** The codata type family $\mathsf{Tri} : \mathsf{Type} \to \mathsf{Setoid}$ of Example 10 is equipped with a structure of a comonad relative to the functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ with $\mathsf{counit}_A := \mathsf{top}_A$ and $\mathsf{cobind}_{A,B} := \mathsf{redec}_{A,B}$.

▶ **Remark 21.** *A* weak constructive comonad *as defined by Matthes and Picard [18] to characterize the codata type* Tri *and redecoration on it, is* precisely *a comonad relative to the functor* $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$*.*

▶ **Remark 22** (Comonads into cartesian closed categories)**.** *With the notations of Definition 12, suppose that the category $\mathcal{D}$ is cartesian closed, i.e. that $\mathcal{D}$ has internal hom-objects. This is*

the case, e.g., for the category Setoid *of setoids. The cobind operation of a relative comonad into $\mathcal{D}$ might then be defined to be a family of arrows in $\mathcal{D}$,*

$$\mathsf{cobind}_{A,B} : \underline{\mathcal{D}}(TA, FB) \to \underline{\mathcal{D}}(TA, TB) \ . \tag{5}$$

The comonads Stream *(cf. Example 17) and* Tri *(cf. Example 19) are comonads with such a cobind operation. In these two cases, this "enriched" definition of the cobind operation given in Equation* (5) *encodes a higher-order compatibility of cosubstitution with bisimilarity, namely that cosubstitution with two extensionally bisimilar functions yields extensionally bisimilar functions.*

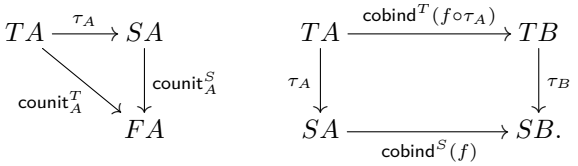
The notion of relative comonad captures many properties of Stream resp. Tri and cosubstitution on them, in particular the interplay of cosubstitution with the destructors head resp. top via the first two axioms. In order to capture the interplay of cosubstitution with the destructor tail (for streams) resp. rest (for infinite triangular matrices), we develop the notion of *comodule over a relative comonad* and, more importantly, their morphisms, in Section 5.

Morphisms of relative comonads are families of morphisms that are compatible with the comonadic structure:

▶ **Definition 23.** Let T and S be comonads relative to a functor $F : \mathcal{C} \to \mathcal{D}$. A **morphism of relative comonads** $\tau : T \to S$ is given by a family of morphisms

$$\tau_A : TA \to SA$$

such that the following diagrams commute for any $A, B : \mathcal{C}_0$ and $f : SA \to FB$:



Relative comonads over a fixed functor F and their morphisms form a category $\mathsf{RComon}(F)$ with the identity and composition operations given pointwise.

▶ **Remark 24.** *A morphism $\tau : T \to S$ of relative comonads over a functor $F : \mathcal{C} \to \mathcal{D}$ is* natural *with respect to the functorial action of Definition 13.*

▶ **Example 25** (Example 15 continued)**.** Let M and M' be two comonads on $\mathcal{C}$, and let $\tau : M \to M'$ be a comonad morphism. The family of morphisms

$$F\tau_A := F(\tau_A) : FMA \to FM'A$$

constitutes a morphism of relative comonads $F\tau : FM \to FM'$.

▶ **Remark 26.** *The definitions given in Examples 15 and 25 yield a functor from comonads on $\mathcal{C}$ to comonads relative to $F : \mathcal{C} \to \mathcal{D}$. If F is a right adjoint with left adjoint L, $L \dashv F$, then postcomposing a comonad T relative to F with the functor L yields a monad on $\mathcal{C}$. Again, this map extends to morphisms. The two functors between categories of monads thus defined are again adjoints. Writing down the details is lengthy but easy.*

An example of comonad morphism is given by the *diagonal* map which extracts the diagonal of an infinite triangular matrix (over type A), yielding a stream over A:

▶ **Example 27.** We define a morphism of relative comonads $\mathsf{diag} : \mathsf{Tri} \to \mathsf{Stream}$ as follows: given a matrix $t : \mathsf{Tri}A$, its diagonal is a stream $\mathsf{diag}_A\ t : \mathsf{Stream}A$. The map diag_A is defined via the clauses

$$\mathsf{head} \circ \mathsf{diag}_A := \mathsf{top} \quad \text{and} \quad \mathsf{tail} \circ \mathsf{diag}_A := \mathsf{diag}_A \circ \mathsf{cut} \circ \mathsf{rest}\ .$$

Instead of proving the comonad morphism axioms for this map, we will, in Example 45, specify the same map via a universal property. There, the compatibility of this map with cosubstitution on source and target will follow for free.

▶ **Remark 28** (Non-examples)**.** *The family of destructors* $\mathsf{tail}_A : \mathsf{Stream}A \to \mathsf{Stream}A$ *does* not *satisfy the axioms for a morphism of relative comonads* $\mathsf{Stream} \to \mathsf{Stream}$.

Similarly, while the map $A \mapsto \mathsf{Tri}(E \times A)$ *inherits the structure of a relative comonad (see Definition 29), the family* $\mathsf{rest}_A : \mathsf{Tri}A \to \mathsf{Tri}(E \times A)$ *does* not *constitute a comonad morphism of type* $\mathsf{Tri} \to \mathsf{Tri}(E \times \_)$.

Let $\mathcal{C}$ be a category with products and $E : \mathcal{C}_0$ be an object of $\mathcal{C}$. The following definition shows how a relative comonad T with domain category $\mathcal{C}$ gives rise to a relative comonad with underlying object map $A \mapsto T(E \times A)$. We shall not make use of this definition in what follows: indeed, in Section 6 we consider the map $A \mapsto T(E \times A)$ as a *comodule* over T, rather than a comonad. The below definition may thus be skipped by the reader.

▶ **Definition 29.** Let T be a comonad relative to a product-preserving functor $F : \mathcal{C} \to \mathcal{D}$ between categories with finite products, and let $E : \mathcal{C}_0$ be a fixed object of $\mathcal{C}$. The map $A \mapsto T(E \times A)$ inherits the structure of a comonad relative to F from T: the counit is defined as

$$\mathsf{counit}_A := \mathsf{counit}^T_A \circ T(\mathsf{pr}_2(E, A))$$

and the cobind operation as

$$\begin{aligned}\mathsf{cobind}_{A,B} : \mathcal{D}\big(T(E \times A), FB\big) &\to \mathcal{D}\big(T(E \times A), T(E \times B)\big)\\ f &\mapsto \mathsf{cobind}^T(\mathsf{lift}'\ f)\end{aligned}$$

with lift' defined as

$$\begin{aligned}\mathsf{lift}' : \mathcal{D}\big(T(E \times A), FB\big) &\to \mathcal{D}\big(T(E \times A), F(E \times B)\big)\ ,\\ f &\mapsto {\phi^F_{E,B}}^{-1} \circ \langle \mathsf{counit}^T_E \circ T(\mathsf{pr}_1), f\rangle\ .\end{aligned}$$

5 Comodules over relative comonads

In this section we develop the notion of *comodule over a relative comonad*, dualizing the notion of module over a relative monad [3]. Our motivation for developing this notion is the characterization of the destructors tail and rest as *morphisms of comodules*.

▶ **Definition 30.** Let T be a comonad relative to $F : \mathcal{C} \to \mathcal{D}$, and let $\mathcal{E}$ be a category. A **comodule over T towards** $\mathcal{E}$ consists of

- a map $M : \mathcal{C}_0 \to \mathcal{E}_0$ on the objects of the categories involved and
- an operation

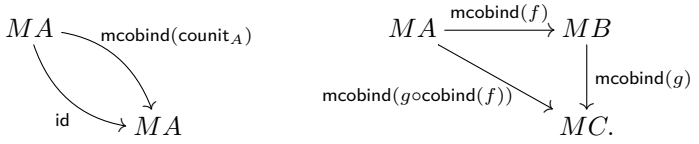
$$\mathsf{mcobind} : \forall A, B : \mathcal{C}_0, \mathcal{D}(TA, FB) \to \mathcal{E}(MA, MB)$$

such that the following equations hold:

$$\forall A : \mathcal{C}_0, \mathsf{mcobind}(\mathsf{counit}_A) = \mathsf{id}_{MA} \tag{6}$$

$$\forall A, B, C : \mathcal{C}_0, \forall f : \mathcal{D}(TA, FB), \forall g : \mathcal{D}(TB, FC), \\ \mathsf{mcobind}(g) \circ \mathsf{mcobind}(f) = \mathsf{mcobind}(g \circ \mathsf{cobind}(f)) \ , \tag{7}$$

in diagrammatic form:



Similarly to relative comonads, comodules over these are functorial:

▶ **Definition 31** (Functoriality for comodules)**.** Let $M : \mathsf{RComod}(T, \mathcal{E})$ be a comodule over T towards some category $\mathcal{E}$. For $f : \mathcal{C}(A, B)$ we define

$$M(f) := \mathsf{mcobind}(Ff \circ \mathsf{counit}_A) \ .$$

Every relative comonad constitutes a comodule over itself, the *tautological* comodule:

▶ **Definition 32** (Tautological comodule)**.** Given a comonad T relative to $F : \mathcal{C} \to \mathcal{D}$, the map $A \mapsto TA$ yields a comodule over T with target category $\mathcal{D}$, the **tautological comodule** of T, also called T. The comodule operation is given by $\mathsf{mcobind}^T(f) := \mathsf{cobind}^T(f)$.

A more interesting example of comodule is given by the functor that maps a type A to the setoid $\mathsf{Tri}(E \times A)$ for some fixed type E:

▶ **Example 33.** The map $A \mapsto \mathsf{Tri}(E \times A)$ is equipped with a comodule structure over the relative comonad Tri by defining the comodule operation $\mathsf{mcobind}$ as (cf. Example 10)

$$\mathsf{mcobind}_{A,B}(f) := \mathsf{redec}_{E\times A, E\times B}(\mathsf{lift}\ f) \quad \text{for} \quad f : \mathsf{Setoid}(\mathsf{Tri}A, \mathsf{eq}B) \ .$$

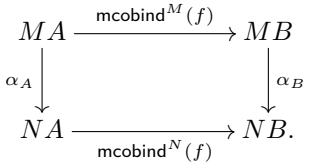
In Section 6 we generalize Example 33 – precomposition with a product – to more general relative comonads over suitable categories. However, this requires an axiomatization of the lift operation, more precisely of an important building block of it, the cut operation.

A *morphism of comodules* is given by a family of morphisms that is compatible with the comodule operation:

▶ **Definition 34** (Morphism of comodules)**.** Let M and $N : \mathcal{C} \to \mathcal{E}$ be comodules over the comonad T relative to $F : \mathcal{C} \to \mathcal{D}$. A **morphism of comodules** from M to N is given by a family of morphisms $\alpha_A : \mathcal{E}(MA, NA)$ such that for any $A, B : \mathcal{C}_0$ and $f : \mathcal{D}(TA, FB)$ one has

$$\alpha_B \circ \mathsf{mcobind}^M(f) = \mathsf{mcobind}^N(f) \circ \alpha_A \ ,$$

in diagrammatic form



Composition and identity of comodule morphisms happens pointwise. We thus obtain a category $\mathsf{RComod}(T, \mathcal{E})$ of comodules over a fixed comonad T, towards a fixed category $\mathcal{E}$.

The motivating examples for us to consider comodules and their morphisms are the following:

▶ **Example 35.** The family of destructors $\mathsf{tail}_A : \mathsf{Stream}A \to \mathsf{Stream}A$, indexed by a type A, is the carrier of a morphism of tautological comodules (over the relative comonad Stream),

$$\mathsf{tail} : \mathsf{Stream} \to \mathsf{Stream} \ .$$

▶ **Example 36.** The family of destructors $\mathsf{rest}_A : \mathsf{Tri}A \to \mathsf{Tri}(E \times A)$, indexed by a type A, of Example 10 is a morphism of comodules over the comonad Tri from the tautological comodule Tri to the comodule $\mathsf{Tri}(E \times \_)$ as defined in Example 33,

$$\mathsf{rest} : \mathsf{Tri} \to \mathsf{Tri}(E \times \_) \ .$$

▶ **Remark 37.** *The family of morphisms constituting a comodule morphism is actually natural with respect to the functoriality defined in Definition 31.*

Given a morphism of comonads, we can "transport" comodules over the source comonad to comodules over the target comonad:

▶ **Definition 38** (Pushforward comodule)**.** Let $\tau : T \to S$ be a morphism of comonads relative to a functor $F : \mathcal{C} \to \mathcal{D}$, and let furthermore M be a comodule over T towards a category $\mathcal{E}$. We define the **pushforward comodule** $\tau_* M$ to be the comodule over S given by $\tau_* M(A) := MA$ and, for $f : \mathcal{D}(SA, FB)$,

$$\mathsf{mcobind}^{\tau_* M}(f) := \mathsf{mcobind}^{M}(f \circ \tau_A) : \mathcal{E}(MA, MB) \ .$$

Pushforward is functorial: if M and N are comodules over T with codomain category $\mathcal{E}$, and $\alpha : M \to N$ is a morphism of comodules, then we define $\tau_* \alpha : \tau_* M \to \tau_* N$ as the family of morphisms $(\tau_* \alpha)_A := \alpha_A$. It is easy to check that this is a morphism of comodules (over S) between $\tau_* M$ and $\tau_* N$. Pushforward thus yields a functor

$$\tau_* : \mathsf{RComod}(T, \mathcal{E}) \to \mathsf{RComod}(S, \mathcal{E}) \ .$$

As presented in Definition 32, every relative comonad induces a comodule over itself. This extends to morphisms of relative comonads in the following sense:

▶ **Definition 39.** Let $\tau : T \to S$ be a morphism of comonads relative to $F : \mathcal{C} \to \mathcal{D}$. Then τ gives rise to a morphism of comodules over S from the pushforward of the tautological comodule of T along τ to the tautological comodule over S,

$$\langle \tau \rangle : \tau_* T \to S \ , \quad \langle \tau \rangle_A := \tau_A \ .$$

We conclude the section with some constructions of comodules:

▶ **Lemma 40.** *Let T be a comonad relative to $F : \mathcal{C} \to \mathcal{D}$, and let $M : \mathcal{C} \to \mathcal{E}$ be a comodule over T. Let $G : \mathcal{E} \to \mathcal{X}$ be a functor. Then $(G \circ M, G \circ \mathsf{mcobind}^{M})$ is a comodule over T. This extends to morphisms of comodules: for N another comodule over T with target category $\mathcal{E}$, and $\alpha : M \to N$ a morphism of comodules, $G \circ \alpha, c \mapsto G(\alpha_c)$ is a morphism of comodules from $G \circ M$ to $G \circ N$. This yields a functor $\mathsf{RComod}(T, \mathcal{E}) \to \mathsf{RComod}(T, \mathcal{X})$.*

In particular, given any object $e : \mathcal{E}_0$, the constant functor $\mathcal{C} \to \mathcal{E}$ mapping to e is a comodule for any comonad relative to $F : \mathcal{C} \to \mathcal{D}$.

▶ **Lemma 41.** *Let T be a comonad relative to $F : \mathcal{C} \to \mathcal{D}$, and let $M, N : \mathcal{C} \to \mathcal{E}$ be comodules over T. Suppose that $\mathcal{E}$ has coproducts, denoted $e + e'$. Then $M + N, c \mapsto Mc + Nc$ is a comodule over T, with cosubstitution given by* $\mathsf{mcobind}^M + \mathsf{mcobind}^N$. *This construction extends to a coproduct on* $\mathsf{RComod}(T, \mathcal{E})$.

6 Terminality for streams and infinite triangular matrices

In this section, we define a notion of "model" for the signatures of streams and triangular matrices, respectively. We then show that the codata types Stream and Tri constitute the terminal object in the respective category of models.

This terminal semantics result is hardly surprising; however, it is still interesting as it characterizes not only the codata types themselves, but also the respective bisimilarity relations and comonadic operations on them, via a universal property.

6.1 Models for Stream

We first consider the homogeneous codata type of streams.

▶ **Definition 42** (Category of models for Stream)**.** A **model for** Stream is given by a pair (S, t) consisting of

- a comonad S relative to $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ and
- a morphism t of tautological comodules over S,

$$t : S \to S \ .$$

A **morphism of models** $(S, t) \to (S', t')$ is given by a comonad morphism $\tau : S \to S'$ such that $\langle \tau \rangle \circ \tau_* t = t' \circ \langle \tau \rangle$, in diagrammatic form,

$$\begin{array}{ccc} S & \xrightarrow{\tau_* t} & S \\ {\scriptstyle \langle \tau \rangle}\downarrow & & \downarrow{\scriptstyle \langle \tau \rangle} \\ S' & \xrightarrow[t']{} & S'. \end{array}$$

Note that the above diagram is a diagram in the category $\mathsf{RComod}(S', \mathsf{Setoid})$.

This defines a category of models of the signature of streams, with composition and identity inherited from those of comonad morphisms.

In the introduction we mentioned a more traditional notion of "model" for a signature of a coinductive data type, namely that of a coalgebra of some endofunctor. The relationship between models as in Definition 42 and coalgebras is as follows:

▶ **Remark 43** (Coalgebras for streams: forgetting cosubstitution)**.** *Define the functor*

$$F : [\mathsf{Type}, \mathsf{Setoid}] \to [\mathsf{Type}, \mathsf{Setoid}] \ , \quad F(G)(A) := \mathsf{eq}(A) \times G(A) \ .$$

Then we have a forgetful functor from models for streams (in the sense of Definition 42) to coalgebras of F which, intuitively, forgets the cosubstitution structure. Indeed, given a model (S, t), the comonad S is, in particular, a functor $S : \mathsf{Type} \to \mathsf{Setoid}$ which constitutes the carrier of a coalgebra of F: the coalgebra structure map $S \to F(S) := \mathsf{eq} \times S$, as a map into a product, is assembled from the counit of S as the first component, and the comodule morphism $t : S \to S$, which is a natural transformation $t : S \to S$ as the second component.

The reason why we consider the richer notion of model (as compared to coalgebras) is that we want the objects of our category – and thus in particular the terminal object – to be equipped with a well-behaved "cosubstitution" operation. It is this cosubstitution operation which is modeled by the comonad and comodule structure, and which is forgotten by the forgetful functor defined above.

The category of models for Stream has a terminal object:

▶ **Theorem 44.** *The pair* $(\mathsf{Stream}, \mathsf{tail})$*, where* Stream *is considered as a relative comonad and* tail *as a morphism of comodules, is the terminal object in the category of models of Definition 42.*

More precisely, Theorem 44 says that the rules given in Figure 2 allow to prove that the category of models defined in Definition 42 has a terminal object. The proof of Theorem 44 being essentially the same as that of Theorem 56, we omit the former. However, a mechanized proof can be found in our Coq library, see Section 7.

The property of being terminal can be used to specify a map into streams, by equipping some comonad S relative to $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ with the structure of a model for streams, that is, with a comodule morphism $S \to S$. We do so for the comonad Tri:

▶ **Example 45.** We equip the relative comonad Tri with the structure of a model for Stream by defining a morphism of tautological comodules over Tri, given by the composition

$$t^{\mathsf{diag}} := \mathsf{cut} \circ \mathsf{rest} : \mathsf{Tri} \to \mathsf{Tri} \ .$$

By terminality we obtain a morphism of models

$$(\mathsf{Tri}, t^{\mathsf{diag}}) \to (\mathsf{Stream}, \mathsf{tail})$$

which has, as underlying morphism of relative comonads, the one defined in Example 27. Compatibility of this map with cosubstitution in Tri and Stream, respectively, is for free.

▶ **Remark 46.** *Fix a type B. A result analogous to Theorem 44 holds for trees* Tree_B *of Example 19. We refrain from giving a precise statement of this result.*

6.2 Models for Tri

In analogy to the definition of models for the signature of streams, one would like to define a model for the signature of Tri as a pair (T, r) of a comonad T relative to $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ and a morphism of comodules $r : T \to T(E \times \_)$. It turns out that in this way, one does not obtain the right auxiliary function cut for what is supposed to be the *terminal* such model, the pair $(\mathsf{Tri}, \mathsf{rest})$, where cut is used to define the comodule $\mathsf{Tri}(E \times \_)$. As a remedy, we define a model to come equipped with a specified operation analogous to cut, and some laws governing the behavior of that operation:

▶ **Definition 47.** Let $\mathcal{C}$ and $\mathcal{D}$ be categories with binary products and $F : \mathcal{C} \to \mathcal{D}$ a product-preserving functor. Let $E : \mathcal{C}_0$ be a fixed object of $\mathcal{C}$. We define a **comonad relative to** F **with cut relative to** E to be a comonad T relative to F together with a cut operation

$$\mathsf{cut} : \forall\ A : \mathcal{C}_0, T(E \times A) \to TA$$

satisfying the axioms

- $\forall\ A : C_0, \mathsf{counit}_A \circ \mathsf{cut}_A = \mathsf{counit}_A \circ T(\mathsf{pr}_2(E, A))$;
- $\forall\ A\ B : C_0, \forall\ f : \mathcal{D}(TA, FB), \mathsf{cobind}(f) \circ \mathsf{cut}_A = \mathsf{cut}_B \circ \mathsf{cobind}(\mathsf{lift}\ f)$,

that is,

$$T(E \times A) \underset{\mathsf{cut}_A}{\overset{T(\mathsf{pr}_2(E,A))}{\rightrightarrows}} TA \xrightarrow{\mathsf{counit}_A} FA \qquad \text{and} \qquad \begin{array}{ccc} T(E \times A) & \xrightarrow{\mathsf{cobind}(\mathsf{lift}\ f)} & T(E \times B) \\ {\scriptstyle \mathsf{cut}_A}\downarrow & & \downarrow{\scriptstyle \mathsf{cut}_B} \\ TA & \xrightarrow[\mathsf{cobind}(f)]{} & TB \end{array}$$

where, for $f : \mathcal{D}(TA, FB)$, we define $\mathsf{lift}(f) : \mathcal{D}\big(T(E \times A), F(E \times B)\big)$ as

$$\mathsf{lift}(f) := {\phi^F_{E,B}}^{-1} \circ (\mathsf{counit}_E \times f) \circ \langle T(\mathsf{pr}_1), \mathsf{cut} \rangle \ .$$

Morphisms of comonads with cut are morphisms of comonads that are compatible with the respective cut operations:

▶ **Definition 48.** Let (T, cut^T) and (S, cut^S) be two comonads relative to a functor F with cut relative to E as in Definition 47. A **morphism of comonads with cut** is a comonad morphism τ between the underlying comonads as in Definition 23 that commutes suitably with the respective cut operations, i.e. for any $A : \mathcal{C}_0$, $\mathsf{cut}^S_A \circ \tau_{E \times A} = \tau_A \circ \mathsf{cut}^T_A$:

$$\begin{array}{ccc} T(E \times A) & \xrightarrow{\mathsf{cut}^T_A} & TA \\ {\scriptstyle \tau_{E\times A}}\downarrow & & \downarrow{\scriptstyle \tau_A} \\ S(E \times A) & \xrightarrow[\mathsf{cut}^S_A]{} & SA. \end{array}$$

Comonads with cut relative to a fixed functor $F : \mathcal{C} \to \mathcal{D}$ and $E : \mathcal{C}_0$ form a category $\mathsf{RComonwCut}(F, E)$. There is a forgetful functor from $\mathsf{RComonwCut}(F, E)$ to $\mathsf{RComon}(F)$. Conversely, any comonad T relative to a suitable functor can be equipped with a cut operation, using functoriality of T.

▶ **Remark 49** (Canonical cut operation)**.** *Any comonad T relative to a product-preserving functor $F : \mathcal{C} \to \mathcal{D}$ can be equipped with a* cut *operation relative to $E : \mathcal{C}_0$ satisfying the properties of Definition 47 by setting $\mathsf{ccut}_A := \mathsf{cut}_A := T\big(\mathsf{pr}_2(E, A)\big)$. (The extra "c" of* ccut *stands for "canonical".) It follows from the axioms of comonad morphisms that a comonad morphism $\tau : T \to S$ satisfies the equation of Definition 48 for the operations ccut^T and ccut^S thus defined. The morphism τ hence constitutes a morphism of comonads with cut from (T, ccut^T) to (S, ccut^S). We thus obtain a functor*

$$\mathsf{ccut}_{F,E} : \mathsf{RComon}(F) \to \mathsf{RComonwCut}(F, E)$$

from relative comonads over F to relative comonads over F with cut relative to a fixed object $E : \mathcal{C}_0$ given on objects by $T \mapsto (T, \mathsf{ccut}^T)$.

The functor $\mathsf{ccut}_{F,E}$, followed by the forgetful functor, yields the identity. We can thus view relative comonads with cut as a generalization of relative comonads.

Our prime example of relative comonad comes with a cut operation that is *not* the canonical one:

▶ **Example 50.** The relative comonad Tri from Example 20, together with the cut operation defined in Example 10, is a comonad with cut as in Definition 47.

Given a comodule M over a relative comonad T with cut, we define a comodule over T obtained by precomposition of M with "product with a fixed object E":

▶ **Definition 51.** Suppose $F : \mathcal{C} \to \mathcal{D}$ is a product-preserving functor, and T is a comonad relative to F with a cut operation relative to $E : \mathcal{C}_0$ as in Definition 47. Given a comodule M over T, **precomposition with** "**product** with E" gives a comodule

$$M(E \times \_) : A \mapsto M(E \times A)$$

over T. The comodule operation is induced by that of M by

$$\mathsf{mcobind}^{M(E\times\_)}_{A,B} : \mathcal{D}(TA, FB) \to \mathcal{E}\big(M(E \times A), M(E \times B)\big) \ ,$$
$$f \mapsto \mathsf{mcobind}^{M}_{E\times A, E\times B}(\mathsf{lift}(f)) \ ,$$

where the lift operation is the one defined in Definition 47.

Furthermore, given two comodules M and N over $\mathcal{T}$ with target category $\mathcal{E}$, and a comodule morphism $\alpha : M \to N$, the assignment $\alpha(E \times \_)_A := \alpha_{E\times A}$ defines a comodule morphism $\alpha(E \times \_) : M(E \times \_) \to N(E \times \_)$.
We thus obtain an endofunctor on the category of comodules over T towards $\mathcal{E}$,

$$M \mapsto M(E \times \_) : \mathsf{RComod}(T, \mathcal{E}) \to \mathsf{RComod}(T, \mathcal{E}) \ .$$

▶ **Remark 52** (Pushforward commutes with product in context)**.** *The constructions of Definitions 51 and 38 commute: we have an isomorphism of comodules*

$$\tau_*(M(E \times \_)) \cong (\tau_* M)(E \times \_)$$

given pointwise by identity morphisms.

It directly follows from the definition that the cut operation of any comonad T with cut constitutes a comodule morphism $\mathsf{cut} : T(E \times \_) \to T$. We can thus restate the definition of a morphism of comonads with cut as in Definition 48 by asking the following diagram of comodule morphisms (in the category $\mathsf{RComod}(S, \mathcal{D})$) to commute (where in the upper left corner we silently add an isomorphism as in Remark 52):

$$\begin{array}{ccc} \tau_* T(E \times \_) & \xrightarrow{\tau_*(\mathsf{cut}^T)} & \tau_* T \\ {\scriptstyle \langle\tau\rangle(E\times\_)}\downarrow & & \downarrow{\scriptstyle \langle\tau\rangle} \\ S(E \times \_) & \xrightarrow[\mathsf{cut}^S]{} & S \end{array} \ .$$

The construction of Definition 51 yields a categorical characterization of the rest destructor – more precisely, of its behavior with respect to cosubstitution as in Equation 1 – via the notion of comodule morphism:

▶ **Example 53.** Definition 51 is an axiomatization of Example 36: the relative comonad Tri is a relative comonad with cut, and the family of destructors $\mathsf{rest}_A : \mathsf{Tri}A \to \mathsf{Tri}(E \times A)$ constitutes a morphism of comodules

$$\mathsf{rest} : \mathsf{Tri} \to \mathsf{Tri}(E \times \_)$$

from the tautological comodule Tri to the composed comodule $\mathsf{Tri}(E \times \_)$ (obtained by precomposing the tautological comodule Tri with "product with E" as defined in Definition 51).

The axiomatization of the cut operation now allows us to define models for the signature of infinite triangular matrices:

▶ **Definition 54** (Models for infinite triangular matrices)**.** Let $E : \mathsf{Type}$ be a fixed type. Let $\mathcal{T} = \mathcal{T}_E$ be the category of **models for infinite triangular matrices** where an object is a pair (T, rest^T) consisting of

- a comonad T over the functor $\mathsf{eq} : \mathsf{Type} \to \mathsf{Setoid}$ with cut relative to E and
- a morphism rest^T of comodules over T of type $T \to T(E \times \_)$

such that for any set A, $\mathsf{rest}^T_A \circ \mathsf{cut}^T_A = \mathsf{cut}^T_{E\times A} \circ \mathsf{rest}^T_{E\times A}$.

The last equation can be stated as an equality of comodule morphisms, diagrammatically

$$\begin{array}{ccc} T(E \times \_) & \xrightarrow{\mathsf{rest}^T(E\times\_)} & T(E \times E \times \_) \\ {\scriptstyle \mathsf{cut}^T}\downarrow & & \downarrow{\scriptstyle \mathsf{cut}^T(E\times\_)} \\ T & \xrightarrow[\mathsf{rest}^T]{} & T(E \times \_). \end{array}$$

A morphism between two such objects (T, rest^T) and (S, rest^S) is given by a morphism of relative comonads with cut $\tau : T \to S$ such that the following diagram of comodule morphisms in the category $\mathsf{RComod}(S, \mathcal{E})$ commutes,

$$\begin{array}{ccc} \tau_* T & \xrightarrow{\tau_*(\mathsf{rest}^T)} & \tau_* T(E \times \_) \\ {\scriptstyle \langle\tau\rangle}\downarrow & & \downarrow{\scriptstyle \langle\tau\rangle(E\times\_)} \\ S & \xrightarrow[\mathsf{rest}^S]{} & S(E \times \_) \ . \end{array} \tag{8}$$

Here we silently insert an isomorphism as in Remark 52 in the upper right corner.

Analogously to the signature for streams, there is a forgetful functor from models for triangular matrices to coalgebras of a specific higher-order endofunctor:

▶ **Remark 55** (Coalgebras for Tri)**.** *Let $E : \mathsf{Type}$ be a type. Define the functor*

$$F : [\mathsf{Type}, \mathsf{Setoid}] \to [\mathsf{Type}, \mathsf{Setoid}] \ , \quad F(G)(A) := \mathsf{eq}(A) \times G(E \times A) \ .$$

There is a forgetful functor from models for infinite triangular matrices to coalgebras of F which, intuitively, forgets the comonadic cobind operation and the cut *operation. Indeed, given a model (T, rest^T), the comonad T is, in particular, a functor $T : \mathsf{Type} \to \mathsf{Setoid}$ which constitutes the carrier of a coalgebra of F. The coalgebra structure map on T, being a map $T \to \mathsf{eq} \times T(E \times \_)$ into a product, is assembled from the counit of T (to give the first component) and the comodule morphism $\mathsf{rest}^T : T \to T(E \times \_)$ (to give the second component). Note that both the counit and rest^T are natural transformations.*

▶ **Theorem 56.** *The pair* $(\mathsf{Tri}, \mathsf{rest})$ *consisting of the relative comonad with cut* Tri *of Example 50 together with the morphism of comodules* rest *of Example 36, constitutes the terminal model of triangular matrices.*

Proof. For a given model (T, rest^T), the (terminal) morphism $\bigcirc = \bigcirc_T : T \to \mathsf{Tri}$ is defined via the corecursive equations

$$\mathsf{top} \circ \bigcirc := \mathsf{counit}^T \quad \text{and} \tag{9}$$

$$\mathsf{rest} \circ \bigcirc := \bigcirc \circ \mathsf{rest}^T \ . \tag{10}$$

```
CoInductive Tri A : Type :=
    constr : A -> Tri (E x A) -> Tri A.
CoInductive bisim : forall {A}, Tri A -> Tri A -> Prop :=
    bisim_constr : forall {A} {t t' : Tri A},
         top t = top t' /\ bisim (rest t) (rest t') -> bisim t t'.
```

Figure 8 Definition of Tri and bisimilarity using Coq's CoInductive vernacular.

Using the introduction axiom for bisimilarity we show that the map $\bigcirc$ commutes with cobind and cut operations of the source and target models. We omit these calculations, which can be consulted in the Coq source files.

Note that there is actually no choice in this definition: Equation (9) is forced upon us since we want $\bigcirc$ to constitute a morphism of comonads – the equation directly corresponds to one of the axioms. Equation (10) is forced upon us by the diagram of Equation (8), which a morphism of models has to make commute.

The same argument is used to show, again by use of the introduction rule for bisimilarity, that any two morphisms of models $\tau, \rho : (T, \mathsf{rest}^T) \to (\mathsf{Tri}, \mathsf{rest})$ are equal in the sense of being pointwise bisimilar, thus concluding the proof. ◀

7 Formalization in Coq

All our definitions and theorems are mechanized in the proof assistant Coq [10]. In the formalization, we axiomatize the considered coinductive type families as given by the rules of Figures 2 and 4. In order to ensure consistency, we

1. use Coq *module* types to encapsulate the axioms, implemented via the Axiom vernacular, and
2. instantiate each module type by defining streams and triangular matrices as coinductive types using the CoInductive vernacular, as shown in Figure 8 for the example of Tri.

This shows that the axioms we add to Coq are weaker than the general mechanism of defining coinductive types in Coq via the CoInductive vernacular. Our results are hence axiom-free with respect to the theory implemented in the Coq proof assistant.

The Coq source files are available from the project web site [5]. The correspondences between formalized statements and the statements in this article are given in Table 1.

Prior to version 8.5 of Coq, a duplication of the definition of *setoids* was necessary in order to avoid a universe inconsistency: indeed, the type of setoids is both the target type of a field in the record of categories as well as the type of objects of the category of setoids, which leads to a complicated graph of universe constraints. In order to work around a universe inconsistency, the type of setoids to be used as objects of the category of setoids hence had to be (re)defined later to obtain a sufficiently large universe level.

Version 8.5 of Coq introduces a new, optional, universe polymorphism [21]. In our code repository [5], we provide a version of our code for Coq 8.4pl6 and two versions for Coq 8.5 (beta2 at the time of writing); one where universe polymorphism is not employed, and where hence the aforementioned duplication of code is necessary, and one where polymorphism is used to get rid of that duplication. However, in that latter version, getting our files accepted by Coq then required giving up on canonical structures, which do not seem to play well (yet) with universe polymorphism.

Table 1 Correspondence of informal and formal definitions.

Informal	Reference	Formal
Category		`Category`
Functor		`Functor`
Relative comonad	Def. 12	`RelativeComonad`
Streams as comonad	Ex. 17	`Stream`
Triangular matrices as comonad	Ex. 20	`Tri`
Comodule over comonad	Def. 30	`Comodule`
Tautological comodule (of T)	Def. 32	`tcomod`, notation `<T>`
tail is comodule morphism	Ex. 35	`Tail`
rest is comodule morphism	Ex. 36	`Rest`
Pushforward comodule	Def. 38	`pushforward`
Induced comodule morphism	Def. 39	`induced_morphism`
Models for streams	Def. 42	`Stream`
Stream is terminal	Thm. 44	`StreamTerminal.Terminality`
Relative comonad with cut	Def. 47	`RelativeComonadWithCut`
Precomposition with product	Def. 51	`precomposition_with_product`
Models for triangular matrices	Def. 54	`TriMat`
Tri is terminal	Thm. 56	`TriMatTerminal.Terminality`

8 Conclusion and future work

We have given a categorical semantics for some homogeneous (streams and trees) and heterogeneous (infinite triangular matrices) codata type families, using the notion of relative comonad and comodule over such comonads.

It remains to investigate more general forms of trees, in particular more general forms of heterogeneity than the one we have considered here. This requires a semantic analysis of the conditions that a functor (on the category of types) responsible for heterogeneity needs to satisfy in order to allow the lifting of a (co)substitution rule.

Acknowledgments. We are grateful to André Hirschowitz, Ralph Matthes and Paige North for many helpful discussions. We thank the referees for their suggestions and comments, and the editors for their efforts in organizing these proceedings.

References

1 Andreas Abel, Brigitte Pientka, David Thibodeau, and Anton Setzer. Copatterns: programming infinite structures by observations. In Roberto Giacobazzi and Radhia Cousot, editors, *Principles of Programming Languages*, pages 27–38. ACM, 2013. doi:10.1145/2429069.2429075.

2 Peter Aczel. Galois: A Theory Development Project, 1993. Technical Report for the 1993 Turin meeting on the Representation of Mathematics in Logical Frameworks. `http://www.cs.man.ac.uk/~petera/papers.html`.

3 Benedikt Ahrens. Modules over relative monads for syntax and semantics. *Mathematical Structures in Computer Science*, FirstView:1–35, 2014. doi:10.1017/S0960129514000103.

4 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-Wellfounded Trees in Homotopy Type Theory. In Thorsten Altenkirch, editor, *13th International Conference on Typed*

Lambda Calculi and Applications (TLCA 2015), volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17–30, Dagstuhl, Germany, 2015. Schloss Dagstuhl–Leibniz-Zentrum für Informatik. `https://hott.github.io/M-types/`.

5 Benedikt Ahrens and Régis Spadotti. Terminal semantics for codata types in intensional Martin-Löf type theory: Coq code. `http://benediktahrens.github.io/coinductives/`.

6 Thorsten Altenkirch, James Chapman, and Tarmo Uustalu. Monads need not be endofunctors. In C.-H. Luke Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *LNCS*, pages 297–311. Springer, 2010.

7 Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. Indexed Containers. Unpublished version, `http://www.cs.nott.ac.uk/~txa/publ/jcont.pdf`.

8 Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda terms using generalized inductive types. In Jörg Flum and Mario Rodríguez-Artalejo, editors, *Proceedings of Computer Science Logic*, volume 1683 of *Lecture Notes in Computer Science*, pages 453–468. Springer, 1999. doi:10.1007/3-540-48168-0_32.

9 Steve Awodey, Nicola Gambino, and Kristina Sojakova. Inductive Types in Homotopy Type Theory. In *Proceedings of Symposium on Logic in Computer Science*, pages 95–104. IEEE, 2012. doi:10.1109/LICS.2012.21.

10 The Coq Development Team. The Coq Proof Assistant. `http://coq.inria.fr`, 2015.

11 Thierry Coquand. Infinite objects in type theory. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs 1993*, volume 806 of *Lecture Notes in Computer Science*, pages 62–78. Springer, 1993. doi:10.1007/3-540-58085-9_72.

12 Peter Dybjer. Representing Inductively Defined Sets by Wellorderings in Martin-Löf's Type Theory. *Theor. Comput. Sci.*, 176(1-2):329–335, 1997. doi:10.1016/S0304-3975(96)00145-4.

13 Marcelo Fiore, Gordon Plotkin, and Daniele Turi. Abstract syntax and variable binding. In *Proceedings of the Symposium on Logic in Computer Science*, pages 193–202, Washington, DC, USA, 1999. IEEE Computer Society. doi:10.1109/LICS.1999.782615.

14 André Hirschowitz and Marco Maggesi. Modules over monads and initial semantics. *Inf. Comput.*, 208(5):545–564, 2010. doi:10.1016/j.ic.2009.07.003.

15 Gérard P. Huet and Amokrane Saïbi. Constructive category theory. In Gordon D. Plotkin, Colin Stirling, and Mads Tofte, editors, *Proof, Language, and Interaction, Essays in Honour of Robin Milner*, pages 239–276. The MIT Press, 2000.

16 Bart Jacobs and Jan Rutten. A tutorial on (co) algebras and (co) induction. *Bulletin-European Association for Theoretical Computer Science*, 62:222–259, 1997.

17 Per Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

18 Ralph Matthes and Celia Picard. Verification of redecoration for infinite triangular matrices using coinduction. In Nils Anders Danielsson and Bengt Nordström, editors, *Workshop on Types for Proofs and Programs*, volume 19 of *LIPIcs*, pages 55–69. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2011. doi:10.4230/LIPIcs.TYPES.2011.55.

19 Ieke Moerdijk and Erik Palmgren. Wellfounded trees in categories. *Ann. Pure Appl. Logic*, 104(1-3):189–218, 2000. doi:10.1016/S0168-0072(00)00012-9.

20 John Power. Abstract syntax: Substitution and binders. *Electron. Notes Theor. Comput. Sci.*, 173:3–16, 4 2007. doi:10.1016/j.entcs.2007.02.024.

21 Matthieu Sozeau and Nicolas Tabareau. Universe polymorphism in coq. In Gerwin Klein and Ruben Gamboa, editors, *Proceedings of Interactive Theorem Proving*, volume 8558 of *Lecture Notes in Computer Science*, pages 499–514. Springer, 2014.

22 Miki Tanaka and John Power. A unified category-theoretic formulation of typed binding signatures. In *Proceedings of the workshop on Mechanized reasoning about languages with variable binding*, MERLIN'05, pages 13–24. ACM, 2005. doi:10.1145/1088454.1088457.

23 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

24 Tarmo Uustalu and Varmo Vene. The dual of substitution is redecoration. In Kevin Hammond and Sharon Curtis, editors, *Scottish Functional Programming Workshop*, volume 3 of *Trends in Functional Programming*, pages 99–110. Intellect, 2001.

A Calculus of Constructions with Explicit Subtyping

Ali Assaf[1,2]

1 **INRIA Paris-Rocquencourt, Paris, France**
2 **École Polytechnique, Palaiseau, France**

Abstract

The calculus of constructions can be extended with an infinite hierarchy of universes and cumulative subtyping. Subtyping is usually left implicit in the typing rules. We present an alternative version of the calculus of constructions where subtyping is explicit. We avoid problems related to coercions and dependent types by using the Tarski style of universes and by adding equations to reflect equality.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases type theory, calculus of constructions, universes, cumulativity, subtyping

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.27

1 Introduction

The predicative calculus of inductive constructions (PCIC), the theory behind the Coq proof system [20], contains an infinite hierarchy of predicative universes $\mathsf{Type}_0 : \mathsf{Type}_1 : \mathsf{Type}_2 : \ldots$ and an impredicative universe $\mathsf{Prop} : \mathsf{Type}_1$ for propositions, together with a cumulativity relation:

$$\mathsf{Prop} \subseteq \mathsf{Type}_0 \subseteq \mathsf{Type}_1 \subseteq \mathsf{Type}_2 \subseteq \ldots.$$

Cumulativity gives rise to an asymmetric subtyping relation $\leq$ which is used in the subsumption rule:

$$\frac{\Gamma \vdash M : A \qquad A \leq B}{\Gamma \vdash M : B}\ .$$

Subtyping in Coq is implicit and is handled by the kernel. Type uniqueness does not hold, as a term can have many non-equivalent types, but a notion of *minimal type* can be defined. While subject reduction does hold, the minimal type of a term is not preserved during reduction.

The goal of this paper is to investigate whether it is possible to make subtyping explicit, by inserting explicit coercions such as

$$\uparrow_0 : \mathsf{Type}_0 \to \mathsf{Type}_1$$

and rely on a kernel that uses only the classic conversion rule:

$$\frac{\Gamma \vdash M : A \qquad A \equiv B}{\Gamma \vdash M : B}\ .$$

In this setting, a well-typed term would have a unique type *up to equivalence* and the type would be preserved during reduction.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 27–46

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

Coercions and dependent types

In the presence of dependent types, coercions can interfere with type checking because $\uparrow_0 (A) \not\equiv A$. As a result, terms that were well-typed in a system with implicit subtyping become ill-typed after introducing explicit coercions.

▶ **Example 1.1.** In the context

$$\Gamma = (a : \mathsf{Type}_0, b : \mathsf{Type}_0, f : a \to b, g : \Pi\, c{:}\mathsf{Type}_1.\, c)\,,$$

the term $f\,(g\,a)$ is well-typed and has type b:

$$\Gamma \vdash f\,(g\,a) : b\ .$$

In a system with explicit subtyping, before inserting coercions, this term is not well-typed because a has type Type_0 while g has type $\Pi\, c{:}\mathsf{Type}_1.\, c$ and $\mathsf{Type}_0 \not\equiv \mathsf{Type}_1$. With an explicit coercion $\uparrow_0$: $\mathsf{Type}_0 \to \mathsf{Type}_1$, the term $g\,(\uparrow_0 (a))$ has type $\uparrow_0 (a)$ but $f\,(g\,(\uparrow_0 (a)))$ is not well-typed because f has type $a \to b$ and $\uparrow_0 (a) \not\equiv a$.

The easiest way to circumvent this problem is to add a new equation

$$\uparrow_0 (A) \equiv A.$$

In other words, we erase the coercions to check if two terms are equivalent. While this solution is straightforward, it unfortunately means that we use ill-typed terms. In a system where equivalence is defined by reduction rules, this solution amounts to adding a new reduction rule $\uparrow_0 (A) \longrightarrow A$, which would completely break subject reduction. A calculus of constructions with explicit subtyping should therefore avoid these rules. The solution to this problem is to use universes *à la Tarski*.

Russell vs. Tarski

There are two ways of presenting universes: the *Russell style* and the *Tarski style*. The first is implicit and is used in the calculus of constructions and in pure type systems [1]. The second is explicit and is mainly used in Martin-Löf's intuitionistic type theory [15]. While the Tarski style is usually regarded as the more fundamental of the two, the Russell style is often used as a practical informal version of the other.

In the Tarski style, we make the distinction between terms and types. Every sort Type_i has a corresponding universe symbol U_i and a decoding function T_i. If A is a term of type U_i, it is not itself a type but a code that represents a type, and $\mathsf{T}_i\,(A)$ is its corresponding type. Types do not have a type and there is a separate judgment for well-formed types. For example, $\pi_i\, x : A.B$ is the term of type U_i that represent the product type $\mathsf{T}_i\,(\pi_i\, x : A.B) \equiv \Pi\, x{:}\mathsf{T}_i\,(A)\,.\ \mathsf{T}_i\,(B)$. In this setting, the context Γ of example 1.1 becomes

$$\Gamma = (a : \mathsf{U}_0, b : \mathsf{U}_0, f : \mathsf{T}_0\,(a) \to \mathsf{T}_0\,(b)\,, g : \Pi\, a{:}\mathsf{U}_1.\ \mathsf{T}_1\,(a))$$

and with the coercion $\uparrow_0$: $\mathsf{U}_0 \to \mathsf{U}_1$, the term $g \uparrow_0 (a)$ has type $\mathsf{T}_1\,(\uparrow_0 (a))$:

$$\Gamma \vdash g\,(\uparrow_0 (a)) : \mathsf{T}_1\,(\uparrow_0 (a))\,.$$

By introducing the following equation at the level of types:

$$\mathsf{T}_1\,(\uparrow_0 (a)) \equiv \mathsf{T}_0\,(a)\,,$$

we get

$$\Gamma \vdash g\,(\uparrow_0\ (a)) : \mathsf{T}_0\,(a)$$

and therefore

$$\Gamma \vdash f\,(g\,(\uparrow_0\ (a))) : \mathsf{T}_0\,(b)\,.$$

Notice that the equation is well-formed because both members are types, not terms, so they only need to both be well-formed.

Using Tarski-style universes allows for a finer and cleaner distinction between terms and types. Aside from solving the problem above, it is better suited for studying the metatheory, e.g. for building models, justifying proof irrelevance and extraction, etc. where the implicit cumulativity would be a pain. It is often considered as the "fundamental" formalization of universes that should be taken as reference, while the Russell style is more convenient to use in practice, e.g. in proof assistants like Coq and Agda.

However, are the two styles equivalent? This question comes up frequently, for example in recent work homotopy type theory [17]. While the question has already been partially answered for intuitionistic type theory, it has never been studied for the calculus of constructions before. The answer turns out to be *no*, the two styles are not always equivalent. Luo [14] already showed that there is some discrepancy between them. Example 1.1 confirms this idea and suggests that explicit subtyping in the Russell style is not possible. More importantly, depending on the system, the Russell style can sometimes be strictly more expressive than the Tarski style because the equality of types is not reflected at the level of terms.

This discrepancy can be adressed in one of 3 ways:

- either discard the Tarski style and justify taking the Russell style as reference,
- or keep things as they are and argue that the difference is acceptable,
- or find a formulation where the two styles are equivalent.

In this paper, we go for the last option by presenting a Tarski version of the cumulative calculus of constructions that is equivalent to the Russell version. The key is to add enough equations to reflect the equality of types at the level of terms.

Reflecting equalities

Within the Tarski style, there are two main ways of introducing universes known as *universes as full reflections* and *universes as uniform constructions* [16]. The first method requires reflecting equalities, meaning that codes corresponding to equivalent types are equivalent: if $\mathsf{T}_i\,(A) \equiv \mathsf{T}_i\,(B)$ then $A \equiv B$. In order to achieve that, additional equations must be introduced such as

$$\uparrow_0\ (\pi_0\, x : A.B) \equiv \pi_1\, x : (\uparrow_0\ (A))\,.\uparrow_0\ (B)\,. \tag{1}$$

The second method drops that principle. Instead, $\uparrow_0$ is used as a constructor to inject types from U_0 into U_1. In practice, the usefulness of reflection has not been shown until now and uniform constructions have been preferred [13, 14, 16].

While reflecting equality can be hard to achieve, we argue here that, on the contrary, it is *essential* to be equivalent to the Russell style. First, we note that a term can have multiple translations with the following example.

▶ **Example 1.2.** With Russell-style universes, in the context $\Gamma = (a : \mathsf{Type}_0, b : \mathsf{Type}_0)$, the term $M = \Pi\, x{:}a.\, b$ has type Type_1:

$$\Gamma \vdash_{\subset} \Pi\, x{:}a.\, b : \mathsf{Type}_1.$$

With Tarski-style universes, this term can be translated in two different ways as $M_1 = \uparrow_0\ (\pi_0\, x : a.b)$ and $M_2 = \pi_1\, x : \uparrow_0\ (a)\,.\uparrow_0\ (b)$:

$$\Gamma \vdash_\uparrow \uparrow_0\ (\pi_0\, x : a.b) : \mathsf{U}_1, \qquad \Gamma \vdash_\uparrow \pi_1\, x : \uparrow_0\ (a)\,.\uparrow_0\ (b) : \mathsf{U}_1.$$

When M_1 and M_2 are used as types, this is not a problem because $\mathsf{T}_1\,(M_1) \equiv \mathsf{T}_1\,(M_2) \equiv \Pi\, x{:}\mathsf{T}_0\,(a)\,.\ \mathsf{T}_0\,(b)$. The problem appears when proving higher-order statements about such terms: if p is an abstract predicate of type $\mathsf{Type}_1 \to \mathsf{Type}_1$ then $\mathsf{T}_1\,(p\,M_1) \not\equiv \mathsf{T}_1\,(p\,M_2)$. As a result, we lose some of the expressivity of the Russell-style universes with implicit subtyping.

▶ **Example 1.3** (Necessity of reflecting equalities)**.** In the context

$$\begin{aligned}\Gamma \quad = \quad & p : \mathsf{Type}_1 \to \mathsf{Type}_1,\\ & q : \mathsf{Type}_1 \to \mathsf{Type}_1,\\ & f : \Pi\, c{:}\mathsf{Type}_0.\ p\,c \to q\,c,\\ & g : \Pi\, a{:}\mathsf{Type}_1.\ \Pi\, b{:}\mathsf{Type}_1.\ p\,(\Pi\, x{:}a.\, b)\\ & a : \mathsf{Type}_0,\\ & b : \mathsf{Type}_0,\end{aligned}$$

the term $f\,(\Pi\, x{:}a.\, b)\ (g\,a\,b)$ has type $q\,(\Pi\, x{:}a.\, b)$:

$$\Gamma \vdash f\,(\Pi\, x{:}a.\, b)\ (g\,a\,b) : q\,(\Pi\, x{:}a.\, b)$$

but the corresponding Tarski-style term

$$f\,(\pi_0\, x : a.b)\ (g{\uparrow_0}\ (a)\ \uparrow_0\ (b))$$

is ill-typed because $\mathsf{T}_1\,(p\,(\pi_1\, x : \uparrow_0\ (a)\,.\uparrow_0\ (b))) \not\equiv \mathsf{T}_1\,(p\,(\uparrow_0\ (\pi_0\, x : a.b)))$. The type corresponding to $q\,(\Pi\, x{:}a.\, b)$ is not provable in the Tarski style without further equations!

Reflecting equality with Equation 1 solves this problem by ensuring that any type has a single term representation up to equivalence. While the equations needed for the predicative universes Type_i have been known for some time [13, 16], the equations for the impredicative universe Prop are less obvious and have not been studied before.

Related work

Geuvers and Wiedijk [6] presented a dependently typed system with explicit conversions. In that system, every conversion is annotated inside the term and there is no implicit conversion rule. Terms have a unique type instead of a unique type *up to equivalence*. To solve the issue of dependent types mentioned above, they rely on an erasure equation similar to $\uparrow_0\ (A) \equiv A$. They also present a variant of the system which does not go through ill-typed terms, but that uses typed heterogeneous equality judgments instead.

In Martin-Löf's intuitionistic type theory, Palmgren [16] and Luo [13] formalized systems with a cumulative hierarchy of predicative universes U_i and an impredicative universe Prop. They both use the Tarski style of universes, which distinguishes between a term A of type U_i and the type $\mathsf{T}_i\,(A)$ that it represents, and which allows them to introduce well-typed equations such as Equation 1. However, they only show how to reflect equality for the predicative universes. As a result, these systems lose some of the expressivity of Russell-style universes with implicit subtyping and are therefore incomplete. Similarly, Herbelin and Spiwack [9] presented a variant of the calculus of constructions with one Type universe and explicit coercions from Prop to Type but they do not reflect equality.

Cousineau and Dowek [5] showed how to embed functional pure type systems in the λΠ-calculus modulo, a logical framework that can be seen as a subset of Martin-Löf's framework where equations are expressed as rewrite rules. When the rewrite system is confluent and strongly normalizing, the λΠ-calculus modulo becomes a decidable version of Martin-Löf's framework. Burel and Boespflug [4] used this embedding to formalize and translate Coq proofs to Dedukti [18], a type-checker based on the λΠ-calculus modulo rewriting, but they handle neither the universe hierarchy nor cumulativity.

Contribution

We present a formulation of the cumulative calculus of constructions where subtyping is explicit. By using Tarski-style universes, we are able to solve the problems related to coercions and dependent types. We show that reflecting equality is necessary for the Tarski style to be equivalent to the Russell style, thus settling an old question for good. Our system fully reflects equality: by introducing additional equations between terms, we ensure that every well-typed term in the original system has a unique representation up to equivalence in the new system.

To our knowledge, this is the first time such work has been done for the cumulative calculus of constructions, which includes both a cumulative hierarchy of predicative universes and an impredicative universe. We also show how to orient the equations into rewrite rules so that equivalence can be defined as a congruence of reduction steps. In summary, this paper answers the question:

What is the system that corresponds to the question mark in Figure 1?

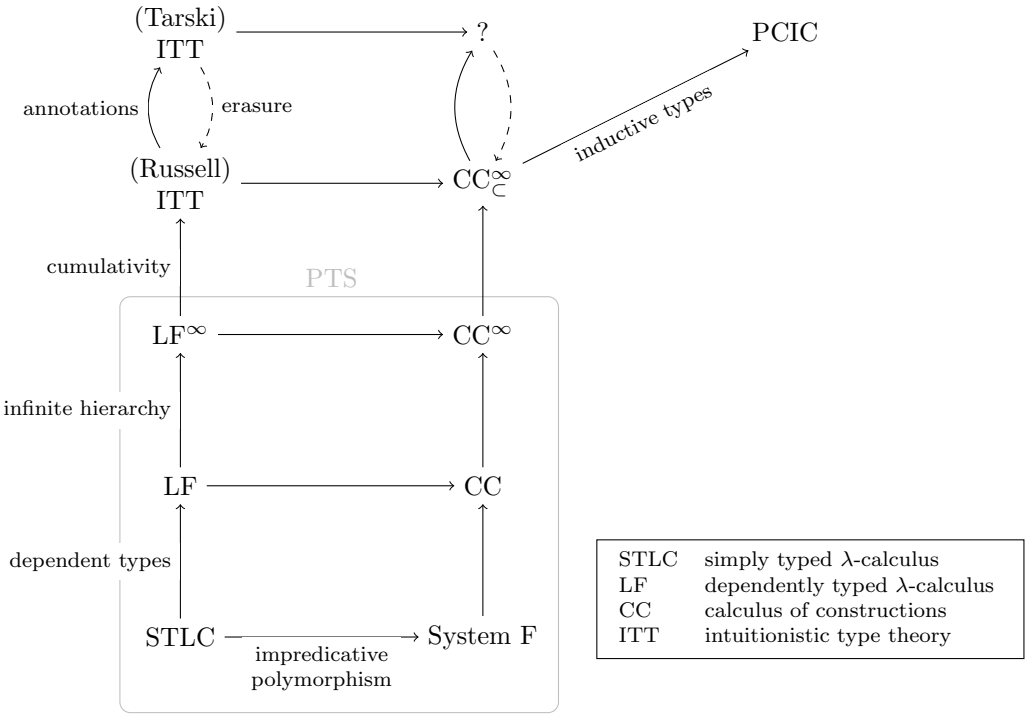


Figure 1 Type theory zoo.

Outline

In Section 2, we present a subset of the original PCIC that we call the *cumulative calculus of constructions* ($\mathrm{CC}^\infty_\subset$). It will serve as the reference to which systems with explicit subtyping will be compared. In Section 3, we present our system with explicit subtyping called the *explicit cumulative calculus of constructions* ($\mathrm{CC}^\infty_\uparrow$). We show exactly which equations are needed to reflect equality. In Section 4, we show that it is complete with respect to $\mathrm{CC}^\infty_\subset$ by defining a translation and proving that it preserves typing. Finally, in Section 5, we show how to transform the equations into rewrite rules so that the system can be implemented in practice.

2 The cumulative calculus of constructions

We consider a subset of PCIC that does not contain inductive types so that we can focus entirely on universes and cumulativity. This system was introduced by Luo [11] under the name $\mathrm{CC}^\infty_\subset$, although with a slightly different presentation. We will also refer to it as the *cumulative calculus of constructions.* It is an extension of the calculus of constructions (CC) with universes and subtyping. It is related to the *extended calculus of constructions* (ECC) [12] but it does not contain sum types. It is also related to the *generalized calculus of constructions* [7, 8] but that one is not fully cumulative as it lacks the $\mathsf{Prop} \subseteq \mathsf{Type}_0$ inclusion[1].

Our presentation differs slightly from Luo's original presentation. The main differences are that $\mathsf{Prop} : \mathsf{Type}_1$ instead of $\mathsf{Prop} : \mathsf{Type}_0$ and that all the rules $(\mathsf{Type}_i, \mathsf{Type}_j, \mathsf{Type}_{\max(i,j)})$ are allowed, as done in Coq. The reason for having Prop in Type_1 instead of Type_0 is mainly historical and not of importance for the purposes of this paper. All of Luo's results still hold for our presentation.

Syntax

The syntax is defined as usual for type theories based on pure type systems. For further background, we refer the reader to [1, 20].

▶ **Definition 2.1** (Syntax).

variables	x, y, α, β	$\in$	$\mathcal{V}$		
sorts	s	$\in$	$\mathcal{S}$	$=$	$\{\mathsf{Prop}\} \cup \{\mathsf{Type}_i \mid i \in \mathbb{N}\}$
terms	M, N, A, B	$\in$	$\mathcal{T}$	$::=$	$x \mid s \mid \Pi\, x{:}A.\, B \mid \lambda\, x{:}A.\, M \mid M\, N$
contexts	Γ, Δ	$\in$	$\mathcal{C}$	$::=$	$. \mid \Gamma, x : A$

Typing

Since the pure type system at the core of $\mathrm{CC}^\infty_\subset$ is functional and complete, we can define its axiom relation $(s_1 : s_2) \in \mathcal{A}$ as a function $\mathcal{A}(s_1)$ and its product rule relation $(s_1, s_2, s_3) \in \mathcal{R}$ as a function $\mathcal{R}(s_1, s_2)$. The cumulativity relation $\subseteq$ can be defined as the reflexive transitive closure of $\mathsf{Prop} \subseteq \mathsf{Type}_0$ and $\mathsf{Type}_i \subseteq \mathsf{Type}_{i+1}$. In order to give a uniform presentation, we define the following operations on sorts.

[1] The $\mathsf{Prop} \subseteq \mathsf{Type}_0$ barrier is often the main difficulty in the metatheory of systems with cumulativity.

▶ **Definition 2.2** (Sort operations)**.** The unary operations $\mathcal{A}$ and $\mathcal{N}$ and the binary operation $\mathcal{R}$ are defined as follows.

$$\begin{array}{lcllcl} \mathcal{A}(\mathsf{Prop}) & = & \mathsf{Type}_1 & \mathcal{R}(\mathsf{Prop}, \mathsf{Prop}) & = & \mathsf{Prop} \\ \mathcal{A}(\mathsf{Type}_i) & = & \mathsf{Type}_{i+1} & \mathcal{R}(\mathsf{Type}_i, \mathsf{Prop}) & = & \mathsf{Prop} \\ \mathcal{N}(\mathsf{Prop}) & = & \mathsf{Type}_0 & \mathcal{R}(\mathsf{Prop}, \mathsf{Type}_j) & = & \mathsf{Type}_j \\ \mathcal{N}(\mathsf{Type}_i) & = & \mathsf{Type}_{i+1} & \mathcal{R}(\mathsf{Type}_i, \mathsf{Type}_j) & = & \mathsf{Type}_{\max(i,j)} \end{array}$$

The cumulativity relation $\subseteq$ is the reflexive transitive closure of $\mathcal{N}$.

To simplify the presentation, we also decouple the context well-formedness judgment from the typing judgment to break the mutual dependency between the two. This formulation is equivalent to the original one but is better suited for proofs by induction. For more information on this common technique, we refer the reader to [21].

▶ **Definition 2.3** (Typing)**.** A term M *has type* A in the context Γ when the judgment $\Gamma \vdash_{\subset} M : A$ can be derived from the following rules.

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_{\subset} x : A}\ \textit{variable}$$

$$\frac{}{\Gamma \vdash_{\subset} s : \mathcal{A}(s)}\ \textit{sort}$$

$$\frac{\Gamma \vdash_{\subset} A : s}{\Gamma \vdash_{\subset} A : \mathcal{N}(s)}\ \textit{cumulativity}$$

$$\frac{\Gamma \vdash_{\subset} A : s_1 \qquad x \notin \Gamma \qquad \Gamma, x : A \vdash_{\subset} B : s_2}{\Gamma \vdash_{\subset} \Pi\, x{:}A.\, B : \mathcal{R}(s_1, s_2)}\ \textit{product}$$

$$\frac{\Gamma \vdash_{\subset} A : s \qquad x \notin \Gamma \qquad \Gamma, x : A \vdash_{\subset} M : B}{\Gamma \vdash_{\subset} \lambda\, x{:}A.\, M : \Pi\, x{:}A.\, B}\ \textit{abstraction}$$

$$\frac{\Gamma \vdash_{\subset} M : \Pi\, x{:}A.\, B \qquad \Gamma \vdash_{\subset} N : A}{\Gamma \vdash_{\subset} M\, N : \{N/x\}B}\ \textit{application}$$

$$\frac{\Gamma \vdash_{\subset} M : A \qquad \Gamma \vdash_{\subset} B : s \qquad A \equiv B}{\Gamma \vdash_{\subset} M : B}\ \textit{conversion}$$

A context Γ *is well-formed* when the judgment $\mathsf{WF}_{\subset}(\Gamma)$ can be derived from the following rules.

$$\frac{}{\mathsf{WF}_{\subset}(.)}\ \textit{empty}$$

$$\frac{\mathsf{WF}_{\subset}(\Gamma) \qquad x \notin \Gamma \qquad \Gamma \vdash_{\subset} A : s}{\mathsf{WF}_{\subset}(\Gamma, x : A)}\ \textit{declaration}$$

We write $\Gamma \vdash M : A$ and $\mathsf{WF}(\Gamma)$ instead of $\Gamma \vdash_{\subset} M : A$ and $\mathsf{WF}_{\subset}(\Gamma)$ when there is no ambiguity.

▶ Remark. The system $\mathrm{CC}^{\infty}_{\subset}$ is *not* equivalent to a non-functional traditional PTS [1]. Indeed, even if the PTS had $\mathsf{Prop} : \mathsf{Type}_i$ for all $i \in \mathbb{N}$ as axioms, the term $\lambda\, a{:}\mathsf{Prop}.\ (\lambda\, a{:}\mathsf{Type}_0.\ a)\ a$ would not be well-typed, while it has type $\mathsf{Prop} \to \mathsf{Type}_0$ with cumulativity in $\mathrm{CC}^{\infty}_{\subset}$.

Unlike in Coq, cumulativity in $\mathrm{CC}^{\infty}_{\subset}$ is not presented as a subtyping rule. This makes it very slightly weaker because product covariance does not hold: if $M : \Pi\, x{:}A.\ \mathsf{Type}_i$ then it does *not* follow that $M : \Pi\, x{:}A.\ \mathsf{Type}_i$. Luo [11] covers this difference in great detail. Since our main focus is the difficulties that arise from universe cumulativity, this is not an issue for us. The notion of subtyping is still useful however. We will define it and use it to caracterize minimal typing.

Minimal typing

The system $\mathrm{CC}^{\infty}_{\subset}$ does not satisfy the uniqueness of types because a term can have multiple non-equivalent types. However, it does have a notion of *minimal type*.

▶ **Definition 2.4** (Subtyping)**.** A term A is a subtype of B when the relation $A \leq B$ can be derived from the following rules where $\equiv$ is the usual β-equivalence relation.

$$\frac{A \equiv B}{A \leq B}\ \textit{reflexivity} \qquad \frac{A \leq B \qquad B \leq C}{A \leq C}\ \textit{transitivity}$$

$$\frac{s_1 \subseteq s_2}{s_1 \leq s_2}\ \textit{cumulativity} \qquad \frac{B \leq C}{\Pi\, x{:}A.\ B \leq \Pi\, x{:}A.\ C}\ \textit{covariance}$$

▶ **Definition 2.5.** A term M *has minimal type* A in the context Γ when $\Gamma \vdash M : A$ and for all B, $\Gamma \vdash M : B$ implies $A \leq B$. We write $\Gamma \vdash_{\mathsf{m}} M : A$.

▶ **Theorem 2.6** (Existence of minimal types)**.** *If* $\Gamma \vdash_{\subset} M : B$ *then there is an* A *such that* $\Gamma \vdash_{\mathsf{m}} M : A$.

Proof. The details can be found in Luo's paper, pages 16–17, [11]. ◀

This notion will be useful when we define our translation. However, note that while minimal types always exist, they are *not* preserved by substitution and β-reduction, as shown in the following example.

▶ **Example 2.7.** In the context $\Gamma = (a : \mathsf{Type}_0, x : \mathsf{Type}_1)$, the term x has minimal type Type_1:

$$a : \mathsf{Type}_0, x : \mathsf{Type}_1 \vdash_{\mathsf{m}} x : \mathsf{Type}_1$$

but the term $\{a/x\}x = a$ has minimal type $\mathsf{Type}_0 \not\equiv \{a/x\}\mathsf{Type}_1$:

$$a : \mathsf{Type}_0 \vdash_{\mathsf{m}} a : \mathsf{Type}_0.$$

3 Explicit subtyping

In this section, we define the *explicit cumulative calculus of constructions* ($\mathrm{CC}^{\infty}_{\uparrow}$) where subtyping is explicit. The syntax is extended to include coercions and to make the distinction between terms and types. We introduce additional equations in the equivalence relation $\equiv$ and give the typing rules based on the rules of $\mathrm{CC}^{\infty}_{\subset}$.

Syntax

For each sort s, we introduce the universe symbol U_s. A term A of type U_s is a code that represents a type in that universe. The decoding function $\mathsf{T}_s\,(A)$ gives the corresponding type. We extend the syntax with the codes u_s and $\pi_{s_1,s_2}\,x : A.B$ that represent the type U_s in the universe $\mathsf{U}_{\mathcal{A}(s)}$ and the product type in the universe $\mathsf{U}_{\mathcal{R}(s_1,s_2)}$ respectively. The universe hierarchy being cumulative, each universe contains codes for all the types of the previous universe using the coercion[2] $\uparrow_s\,(A)$.

▶ **Definition 3.1** (Syntax).

$$\begin{array}{llcll} \text{terms} & M,N,A,B & \in & \mathcal{T} & ::= \quad x \mid \lambda\,x{:}A.\,M \mid M\,N \\ & & & & \quad \mid \mathsf{u}_s \mid \uparrow_s\,(A) \mid \pi_{s_1,s_2}\,x : A.B \\ & & & & \quad \mid \mathsf{U}_s \mid \mathsf{T}_s\,(A) \mid \Pi\,x{:}A.\,B \\ \text{contexts} & \Gamma,\Delta & \in & \mathcal{C} & ::= \quad . \mid \Gamma, x : A \end{array}$$

We write U_i, $\mathsf{T}_i\,(A)$, u_i, $\uparrow_i\,(A)$, and $\pi_{i,j}\,x : A.B$ instead of $\mathsf{U}_{\mathsf{Type}_i}$, $\mathsf{T}_{\mathsf{Type}_i}\,(A)$, $\mathsf{u}_{\mathsf{Type}_i}$, $\uparrow_{\mathsf{Type}_i}\,(A)$, and $\pi_{\mathsf{Type}_i,\mathsf{Type}_j}\,x : A.B$ respectively. When $s_1 \subseteq s_2$ and $s_2 = \mathcal{N}^i(s_1)$, we write $\uparrow_{s_1}^{s_2}\,(A)$ for the term

$$\uparrow_{\mathcal{N}^{i-1}(s)}\,\left(\cdots\uparrow_{\mathcal{N}(s)}\,(\uparrow_s\,(A))\right).$$

For example, $\uparrow_1^3\,(A) = \uparrow_2\,(\uparrow_1\,(A))$ and $\uparrow_{\mathsf{Prop}}^1\,(A) = \uparrow_0\,\left(\uparrow_{\mathsf{Prop}}\,(A)\right)$.

▶ Remark. It is possible to completely split the syntax into distinct categories for types (U_s, $\mathsf{T}_s\,(A)$, $\Pi\,x{:}A.\,B$) and terms. This is one of the advantages of using the Tarski style and it could help simplify the theoretical studies of the calculus of constructions, where the lack of syntactic stratification between terms and types is cumbersome. However, it is not necessary for our purposes so we will not do that here.

Equivalence

Because we are using Tarski-style universes, we need to consider additional equations besides β-equivalence. For now, we just state the equations that are needed and assume a congruence relation $\equiv$ that satisfies those equations. We do not worry about the algorithmic aspect. Later in Section 5, we show how to define $\equiv$ as the usual congruence induced by a set of reduction rules.

In addition to β-equivalence:

$$(\lambda\,x{:}A.\,M)\;N \equiv \{N/x\}M,$$

we need equations to describe the behaviour of the decoding function $\mathsf{T}_s\,(A)$. These are the same as in intuitionistic type theory:

$$\begin{array}{rcl} \mathsf{T}_{\mathcal{A}(s)}\,(\mathsf{u}_s) & \equiv & \mathsf{U}_s \\ \mathsf{T}_{\mathcal{N}(s)}\,(\uparrow_s\,(A)) & \equiv & \mathsf{T}_s\,(A) \\ \mathsf{T}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\,x : A.B) & \equiv & \Pi\,x{:}\mathsf{T}_{s_1}\,(A)\,.\;\mathsf{T}_{s_2}\,(B)\,. \end{array}$$

Finally, we also need equations that reflect equality to ensure that each term of a given type has a unique representation.

[2] One can also view $\uparrow_s\,(A)$ as the code representing $\mathsf{T}_s\,(A)$ in the universe $\mathsf{U}_{\mathcal{N}(s)}$.

Table 1 Different typing derivations for same terms.

$\mathrm{CC}^{\infty}_{\subset}$ typing derivation	$\mathrm{CC}^{\infty}_{\uparrow}$ term representation
$\dfrac{A : \mathsf{Type}_i \qquad x : A \vdash B : \mathsf{Type}_i}{\dfrac{\Pi\, x{:}A.\, B : \mathsf{Type}_i}{\Pi\, x{:}A.\, B : \mathsf{Type}_{i+1}}}$	$\uparrow_i\ (\pi_{i,i}\, x : A.B)$
$\dfrac{\dfrac{A : \mathsf{Type}_i}{A : \mathsf{Type}_{i+1}} \qquad \dfrac{x : A \vdash B : \mathsf{Type}_i}{x : A \vdash B : \mathsf{Type}_{i+1}}}{\Pi\, x{:}A.\, B : \mathsf{Type}_{i+1}}$	$\pi_{i+1,i+1}\, x : \uparrow_i\ (A)\,.\uparrow_i\ (B)$
$\dfrac{A : \mathsf{Type}_i \qquad x : A \vdash B : \mathsf{Prop}}{\Pi\, x{:}A.\, B : \mathsf{Prop}}$	$\pi_{i,\mathsf{Prop}}\, x : A.B$
$\dfrac{\dfrac{A : \mathsf{Type}_i}{A : \mathsf{Type}_{i+1}} \qquad x : A \vdash B : \mathsf{Prop}}{\Pi\, x{:}A.\, B : \mathsf{Prop}}$	$\pi_{i+1,\mathsf{Prop}}\, x : \uparrow_i\ (A)\,.B$
$\dfrac{A : \mathsf{Type}_i \qquad \dfrac{x : A \vdash B : \mathsf{Prop}}{x : A \vdash B : \mathsf{Type}_0}}{\Pi\, x{:}A.\, B : \mathsf{Type}_i}$	$\pi_{i,0}\, x : A.\uparrow_{\mathsf{Prop}}\ (B)$
$\dfrac{\dfrac{\dfrac{A : \mathsf{Type}_i \qquad x : A \vdash B : \mathsf{Prop}}{\Pi\, x{:}A.\, B : \mathsf{Prop}}}{\Pi\, x{:}A.\, B : \mathsf{Type}_0}}{\begin{array}{c}\vdots\\ \Pi\, x{:}A.\, B : \mathsf{Type}_i\end{array}}$	$\uparrow^i_{\mathsf{Prop}}\ (\pi_{i,\mathsf{Prop}}\, x : A.B)$

Which equations are needed to reflect equality? The answer lies in the multiplicity of typing derivations in $\mathrm{CC}^{\infty}_{\subset}$. For example, the product $\Pi\, x{:}A.\, B$ of minimal type Type_0 can be typed at the level Type_1 in two different ways, each giving a different term in $\mathrm{CC}^{\infty}_{\uparrow}$:

$$\dfrac{A : \mathsf{Type}_0 \qquad x : A \vdash B : \mathsf{Type}_0}{\dfrac{\Pi\, x{:}A.\, B : \mathsf{Type}_0}{\Pi\, x{:}A.\, B : \mathsf{Type}_1}} \qquad \uparrow_1\ (\pi_{0,0}\, x : A.B)$$

$$\dfrac{\dfrac{A : \mathsf{Type}_0}{A : \mathsf{Type}_1} \qquad \dfrac{x : A \vdash B : \mathsf{Type}_0}{x : A \vdash B : \mathsf{Type}_1}}{\Pi\, x{:}A.\, B : \mathsf{Type}_1} \qquad \pi_{1,1}\, x : \uparrow_0\ (A)\,.\uparrow_0\ (B)$$

The equivalence relation must therefore take this multiplicity into account. Table 1 lists the different typing derivations that can occur for product types. A careful analysis yields the

following equations:

$$\begin{array}{rcll}
\pi_{\mathcal{N}(s),\mathsf{Prop}}\, x : \uparrow_s\ (A)\,.B & \equiv & \pi_{s,\mathsf{Prop}}\, x : A.B & \\
\pi_{\mathsf{Prop},\mathcal{N}(s)}\, x : A.\uparrow_s\ (B) & \equiv & \uparrow_s\ (\pi_{\mathsf{Prop},s}\, x : A.B) & \\
\pi_{0,j}\, x : \uparrow_{\mathsf{Prop}}\ (A)\,.B & \equiv & \pi_{\mathsf{Prop},j}\, x : A.B & \\
\pi_{i,0}\, x : A.\uparrow_{\mathsf{Prop}}\ (B) & \equiv & \uparrow^{i}_{\mathsf{Prop}}\ (\pi_{i,\mathsf{Prop}}\, x : A.B) & \\
\pi_{i+1,j+1}\, x : \uparrow_i\ (A)\,.B & \equiv & \pi_{i,j+1}\, x : A.B & \text{when } i \leq j \\
\pi_{i+1,j+1}\, x : \uparrow_i\ (A)\,.B & \equiv & \uparrow_i\ (\pi_{i,j+1}\, x : A.B) & \text{when } i > j \\
\pi_{i+1,j+1}\, x : A.\uparrow_j\ (B) & \equiv & \pi_{i+1,j}\, x : A.B & \text{when } i \geq j \\
\pi_{i+1,j+1}\, x : A.\uparrow_j\ (B) & \equiv & \uparrow_j\ (\pi_{i+1,j}\, x : A.B) & \text{when } i < j.
\end{array}$$

It turns out we can express these concisely using the $\uparrow^{s_2}_{s_1}\ (A)$ notation:

$$\begin{array}{rcl}
\pi_{\mathcal{N}(s_1),s_2}\, x : \uparrow_{s_1}\ (A)\,.B & \equiv & \uparrow^{\mathcal{R}(\mathcal{N}(s_1),s_2)}_{\mathcal{R}(s_1,s_2)}\ (\pi_{s_1,s_2}\, x : A.B) \\
\pi_{s_1,\mathcal{N}(s_2)}\, x : A.\uparrow_{s_2}\ (B) & \equiv & \uparrow^{\mathcal{R}(s_1,\mathcal{N}(s_2))}_{\mathcal{R}(s_1,s_2)}\ (\pi_{s_1,s_2}\, x : A.B)\,.
\end{array}$$

▶ **Definition 3.2** (Equivalence). The equivalence relation $\equiv$ is the smallest congruence relation that satisfies the following equations:

$$\begin{array}{rcl}
(\lambda\, x{:}A.\, M)\ N & \equiv & \{N/x\}M \\
\mathsf{T}_{\mathcal{A}(s)}\,(\mathsf{u}_s) & \equiv & \mathsf{U}_s \\
\mathsf{T}_{\mathcal{N}(s)}\,(\uparrow_s\ (A)) & \equiv & \mathsf{T}_s\,(A) \\
\mathsf{T}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\, x : A.B) & \equiv & \Pi\, x{:}\mathsf{T}_{s_1}\,(A)\,.\ \mathsf{T}_{s_2}\,(B) \\
\pi_{\mathcal{N}(s_1),s_2}\, x : \uparrow_{s_1}\ (A)\,.B & \equiv & \uparrow^{\mathcal{R}(\mathcal{N}(s_1),s_2)}_{\mathcal{R}(s_1,s_2)}\ (\pi_{s_1,s_2}\, x : A.B) \\
\pi_{s_1,\mathcal{N}(s_2)}\, x : A.\uparrow_{s_2}\ (B) & \equiv & \uparrow^{\mathcal{R}(s_1,\mathcal{N}(s_2))}_{\mathcal{R}(s_1,s_2)}\ (\pi_{s_1,s_2}\, x : A.B)\,.
\end{array}$$

Typing

To make the distinction between types and terms, we introduce an additional judgment $\Gamma \vdash_\uparrow \mathsf{type}\,(A)$ to capture the property that a type is well-formed. The derivation rules mirror the rules of $\mathrm{CC}^{\infty}_{\subset}$.

▶ **Definition 3.3** (Typing). A term M *has type* A in the context Γ when the judgment $\Gamma \vdash_\uparrow M : A$ can be derived from the following rules, and a term A *is a type* in the context Γ when the judgment $\Gamma \vdash_\uparrow \mathsf{type}\,(A)$ can be derived from the following rules:

$$\frac{(x : A) \in \Gamma}{\Gamma \vdash_\uparrow x : A}\ \textit{variable}$$

$$\frac{}{\Gamma \vdash_\uparrow \mathsf{type}\,(\mathsf{U}_s)}\ \textit{sort-type} \qquad \frac{\Gamma \vdash_\uparrow A : \mathsf{U}_s}{\Gamma \vdash_\uparrow \mathsf{type}\,(\mathsf{T}_s\,(A))}\ \textit{decode-type}$$

$$\frac{\Gamma \vdash_\uparrow \mathsf{type}\,(A) \qquad x \notin \Gamma \qquad \Gamma, x : A \vdash_\uparrow \mathsf{type}\,(B)}{\Gamma \vdash_\uparrow \mathsf{type}\,(\Pi\, x{:}A.\, B)}\ \textit{product-type}$$

$$\frac{}{\Gamma \vdash_\uparrow \mathsf{u}_s : \mathsf{U}_{\mathcal{A}(s)}}\ \textit{sort} \qquad \frac{\Gamma \vdash_\uparrow A : \mathsf{U}_s}{\Gamma \vdash_\uparrow \uparrow_s\ (A) : \mathsf{U}_{\mathcal{N}(s)}}\ \textit{cumulativity}$$

$$\frac{\Gamma \vdash_\uparrow A : \mathsf{U}_{s_1} \qquad x \notin \Gamma \qquad \Gamma, x : \mathsf{T}_{s_1}\,(A) \vdash_\uparrow B : \mathsf{U}_{s_2}}{\Gamma \vdash_\uparrow \pi_{s_1,s_2}\, x : A.B : \mathsf{U}_{\mathcal{R}(s_1,s_2)}}\ \textit{product}$$

$$\frac{\Gamma \vdash_\uparrow \mathsf{type}\,(A) \qquad x \notin \Gamma \qquad \Gamma, x : A \vdash_\uparrow M : B}{\Gamma \vdash_\uparrow \lambda\, x{:}A.\, M : \Pi\, x{:}A.\, B}\ \textit{abstraction}$$

$$\frac{\Gamma \vdash_\uparrow M : \Pi\, x{:}A.\, B \qquad \Gamma \vdash_\uparrow N : A}{\Gamma \vdash_\uparrow M\, N : \{N/x\}B}\ \textit{application}$$

$$\frac{\Gamma \vdash_\uparrow M : A \qquad \Gamma \vdash_\uparrow \mathsf{type}\,(B) \qquad A \equiv B}{\Gamma \vdash_\uparrow M : B}\ \textit{conversion}$$

A context Γ *is well-formed* when the judgment $\mathsf{WF}_\uparrow(\Gamma)$ can be derived from the following rules:

$$\frac{}{\mathsf{WF}_\uparrow(.)}\ \textit{empty} \qquad \frac{\mathsf{WF}_\uparrow(\Gamma) \qquad x \notin \Gamma \qquad \Gamma \vdash_\uparrow \mathsf{type}\,(A)}{\mathsf{WF}_\uparrow(\Gamma, x : A)}\ \textit{declaration}$$

We write $\Gamma \vdash M : A$, $\Gamma \vdash \mathsf{type}\,(A)$, and $\mathsf{WF}\,(\Gamma)$ instead of $\Gamma \vdash_\uparrow M : A$, $\Gamma \vdash_\uparrow \mathsf{type}\,(A)$, and $\mathsf{WF}_\uparrow(\Gamma)$ when there is no ambiguity.

▶ Remark. The equations of Definition 3.2 are well-formed because the left and right side of each equation are either both types or both terms of the same type. In particular, the last two are well-typed because $\mathcal{R}(s_1, s_2) \subseteq \mathcal{R}(\mathcal{N}(s_1), s_2)$ and $\mathcal{R}(s_1, s_2) \subseteq \mathcal{R}(s_1, \mathcal{N}(s_2))$ for all $s_1, s_2 \in \mathcal{S}$.

▶ **Theorem 3.4** (Type uniqueness). *If* $\Gamma \vdash_\uparrow M : A$ *and* $\Gamma \vdash_\uparrow M : B$ *then* $A \equiv B$.

Proof. By induction over the derivations of $\Gamma \vdash_\uparrow M : A$ and $\Gamma \vdash_\uparrow M : B$. We can eliminate conversion rules until we hit a non-conversion rule, in which case we remove the rule from both derivations at the same time. ◀

Erasure

Systems with Tarski-style universes are related to systems with Russell-style universes in a precise sense: we can define an erasure function $|M|$ such that the erasure of a well-typed term in the Tarski style is well-typed in the Russell style. In our setting, this function shows that $\mathrm{CC}^\infty_\uparrow$ is sound with respect to $\mathrm{CC}^\infty_\subset$.

▶ **Definition 3.5** (Erasure). The *term erasure* $|M|$, the *type erasure* $\|A\|$, and the *context erasure* $\|\Gamma\|$ are defined as follows.

$$\begin{array}{lcl}
|x| & = & x \\
|\mathsf{u}_s| & = & s \\
|\uparrow_s (A)| & = & |A| \\
|\pi_{s_1,s_2}\, x : A.B| & = & \Pi\, x{:}|A|\,.\,|B| \\
|\lambda\, x{:}A.\, M| & = & \lambda\, x{:}\|A\|\,.\,|M| \\
|M\, N| & = & |M|\ |N|
\end{array}$$

$$\begin{array}{lcl}
\|\mathsf{U}_s\| & = & s \\
\|\mathsf{T}_s\,(A)\| & = & |A| \\
\|\Pi\, x{:}A.\, B\| & = & \Pi\, x{:}\|A\|\,.\,\|B\|
\end{array}$$

$$\begin{array}{lcl}
\|.\| & = & . \\
\|\Gamma, x : A\| & = & \|\Gamma\|\,, x : \|A\|
\end{array}$$

▶ **Lemma 3.6.** *For all B, x, N, $|\{N/x\}B| = \{|N|/x\}|B|$.*

Proof. By induction on B. ◀

▶ **Theorem 3.7** (Soundness). *If $\Gamma \vdash_\uparrow M : A$ then $\|\Gamma\| \vdash_\subset |M| : \|A\|$. If $\Gamma \vdash_\uparrow \mathsf{type}(A)$ then $\|\Gamma\| \vdash_\subset \|A\| : s$ for some sort s. If $\mathsf{WF}_\uparrow(\Gamma)$ then $\mathsf{WF}_\subset(\|\Gamma\|)$.*

Proof. By induction on the derivations in $\mathrm{CC}^\infty_\uparrow$, using Lemma 3.6 for the application rule. ◀

4 Completeness

In this section, we show that the new system is complete with respect to the original system, meaning that it can express all well-typed terms. We define a function that translates any well-typed term of $\mathrm{CC}^\infty_\subset$ into a term of $\mathrm{CC}^\infty_\uparrow$ and we prove that this translation preserves typing.

Translation

When translating a term, we want to choose the representation that has the minimal type. However, we sometimes need to lift some subterms, such as the argument of applications, in order to get a well-typed term. We therefore define two translations: $[M]_\Gamma$ which translates M according to its minimal type and $[M]_{\Gamma\vdash A}$ which translates M as a term of type A. Finally, since we distinguish between terms and types, we also define $[\![A]\!]_\Gamma$, the translation of A as a type.

▶ **Definition 4.1** (Translation). Let Γ be a well-formed context, A and B be well-formed types in Γ, and M be a well-typed term in Γ such that $\Gamma \vdash_{\mathsf{m}} M : A$ and $\Gamma \vdash_\subset M : B$. The *term translation* $[M]_\Gamma$, the *cast translation* $[M]_{\Gamma\vdash B}$, and the *type translation* $[\![A]\!]_\Gamma$ are mutually defined as follows.

Term translation

$$
\begin{array}{lcl}
[s]_\Gamma & = & \mathsf{u}_s \\
[x]_\Gamma & = & x \\
[\Pi\, x : A'.\, B']_\Gamma & = & \pi_{s_1,s_2}\, x : [A']_\Gamma \,.\, [B']_{\Gamma,x:A'} \\
 & & \text{where } \Gamma \vdash_{\mathsf{m}} A' : s_1 \\
 & & \text{and } \Gamma, x : A' \vdash_{\mathsf{m}} B' : s_2 \\
[\lambda\, x : A'.\, M']_\Gamma & = & \lambda\, x : [\![A']\!]_\Gamma \,.\, [M']_{\Gamma,x:A'} \\
[M'\, N']_\Gamma & = & [M']_\Gamma\, [N']_{\Gamma\vdash A'} \\
 & & \text{where } \Gamma \vdash_{\mathsf{m}} M' : \Pi\, x : A'.\, B'
\end{array}
$$

Cast translation

$$
\begin{array}{lcl}
[M]_{\Gamma\vdash B} & = & [M]_\Gamma \\
 & & \text{when } A \equiv B \\
[M]_{\Gamma\vdash B} & = & \uparrow^{s_2}_{s_1} ([M]_\Gamma) \\
 & & \text{when } A \equiv s_1 \subseteq s_2 \equiv B
\end{array}
$$

Type translation

$$
\begin{array}{lcl}
[\![A]\!]_\Gamma & = & \mathsf{T}_s\, ([A]_\Gamma) \\
 & & \text{where } \Gamma \vdash_{\mathsf{m}} A : s
\end{array}
$$

The context translation $[\![\Gamma]\!]$ where $\mathsf{WF}_{\subset}(\Gamma)$ is defined as follows.

Context translation

$$\begin{array}{lcl} [\![.]\!] & = & . \\ [\![\Gamma, x : A]\!] & = & [\![\Gamma]\!], x : [\![A]\!]_\Gamma \end{array}$$

We will write $[M]$, $[M]_{\vdash C}$, and $[\![A]\!]$ instead of $[M]_\Gamma$, $[M]_{\Gamma\vdash C}$, and $[\![A]\!]_\Gamma$ when unambiguous.

Substitution preservation

A key property for proving the preservation of typing by the translation is that it preserves substitution. If $\Gamma, x : A \vdash_{\subset} M : B$ and $\Gamma \vdash_{\subset} N : A$ then the translation of the substitution is the same as the substitution of the translation. However, the naive statement $\{[N]/x\}[M] \equiv [\{N/x\}M]$ is not true. First, x has type A while N has some minimal type $C \leq A$ so we need to use the cast translation $[N]_{\vdash A}$. Second, the minimal typing is not preserved by substitution, as we showed in Example 2.7. Therefore we also need to fix the type of $\{N/x\}M$ using the cast translation $[\{N/x\}M]_{\vdash\{N/x\}B}$.

▶ **Lemma 4.2** (Translation distributivity). *The translation satisfies the following properties:*

- *For all $s \in \mathcal{S}$, $[\![s]\!] \equiv \mathsf{U}_s$.*
- *If $\Gamma \vdash_{\subset} A : s_1$ and $\Gamma, x : A \vdash_{\subset} B : s_2$ then $[\![\Pi\, x\!:\!A.\, B]\!] \equiv \Pi\, x\!:\![\![A]\!]\,.\,[\![B]\!]$.*
- *If $\Gamma \vdash_{\subset} A : s_1$ and $\Gamma, x : A \vdash_{\subset} B : s_2$ then*

$$[\Pi\, x\!:\!A.\, B]_{\vdash\mathcal{R}(s_1,s_2)} \equiv \pi_{s_1,s_2}\, x : [A]_{\vdash s_1}\,.\,[B]_{\vdash s_2}\,.$$

- *If $\Gamma \vdash_{\subset} A : s_1$ and $\Gamma, x : A \vdash_{\subset} M : B$ then*

$$[\lambda\, x\!:\!A.\, M]_{\vdash\Pi\, y A.\, B} \equiv \lambda\, x\!:\![\![A]\!]\,.\,[M]_{\vdash B}\,.$$

- *If $\Gamma \vdash_{\subset} M : \Pi\, x\!:\!A.\, B$ and $\Gamma \vdash_{\subset} N : A$ then*

$$[M\, N]_{\vdash\{N/x\}B} \equiv [M]_{\vdash\Pi\, x A.\, B}\ [N]_{\vdash A}\,.$$

Proof. Follows from the definition of the equivalence relation $\equiv$ and of the translations $[\![A]\!]$ and $[M]_{\Gamma\vdash A}$. Note that this proposition would not be true if $\equiv$ did not reflect equality. ◀

▶ **Lemma 4.3** (Substitution preservation). *If $\Gamma, x : A, \Gamma' \vdash_{\mathsf{m}} M : B$ and $\Gamma \vdash_{\subset} N : A$ then*

$$\{[N]_{\Gamma\vdash A}/x\}\,[M]_{\Gamma,x:A,\Gamma'} \equiv [\{N/x\}M]_{\Gamma,\{N/x\}\Gamma'\vdash\{N/x\}B}\,.$$

If $\Gamma, x : A, \Gamma' \vdash_{\subseteq} M : B$ and $\Gamma \vdash_{\subset} N : A$ then

$$\{[N]_{\Gamma\vdash A}/x\}\,[M]_{\Gamma,x:A,\Gamma'\vdash B} \equiv [\{N/x\}M]_{\Gamma,\{N/x\}\Gamma'\vdash\{N/x\}B}\,.$$

If $\Gamma, x : A, \Gamma' \vdash_{\subset} M : s$ and $\Gamma \vdash_{\subset} N : A$ then

$$\{[N]_{\Gamma\vdash A}/x\}\,[\![B]\!]_{\Gamma,x:A,\Gamma'} \equiv [\![\{N/x\}B]\!]_{\Gamma,\{N/x\}\Gamma'}\,.$$

Proof. The second and third statements derive from the first. We prove the first by induction on M, using Lemma 4.2.

- Case x. Then we must have $A \equiv B \equiv \{N/x\}B$. Therefore

$$\begin{array}{lcl} \{[N]_{\vdash A}/x\}\,[x] & \equiv & [N]_{\vdash A} \\ & \equiv & [N]_{\vdash\{N/x\}B} \\ & \equiv & [\{N/x\}x]_{\vdash\{N/x\}B}\,. \end{array}$$

- Case $y \neq x$. Then
$$\begin{aligned}\{[N]_{\vdash A}/x\}\,[y] &\equiv y\\ &\equiv [\{N/x\}y]_{\vdash \{N/x\}B}\,.\end{aligned}$$
- Case s. Then
$$\begin{aligned}\{[N]_{\vdash A}/x\}\,[s] &\equiv s\\ &\equiv [\{N/x\}s]_{\vdash \{N/x\}B}\,.\end{aligned}$$
- Case $\Pi\, y{:}C.\, D$. Then $B \equiv s_3$ where $\Gamma, x : A, \Gamma' \vdash_{\mathsf{m}} C : s_1$ and $\Gamma, x : A, \Gamma', y : C \vdash_{\mathsf{m}} D : s_2$ and $s_3 = \mathcal{R}(s_1, s_2)$. Therefore
$$\begin{aligned}\{[N]_{\vdash A}/x\}\,[\Pi\, y{:}C.\, D] &\equiv \pi_{s_1,s_2}\, x : \{[N]_{\vdash A}/x\}\,[C]\,.\{[N]_{\vdash A}/x\}\,[D]\\ &\equiv \pi_{s_1,s_2}\, x : [\{N/x\}C]_{\vdash s_1}\,.\,[\{N/x\}D]_{\vdash s_2}\\ &\equiv [\{N/x\}\,(\Pi\, y{:}C.\, D)]_{\vdash s_3}\end{aligned}$$
- Case $\lambda\, y{:}C.\, M'$. Then $B \equiv \Pi\, x{:}C.\, D$ where $\Gamma, x : A, \Gamma' \vdash_{\mathsf{m}} C : s_1$ and $\Gamma, x : A, \Gamma', y : C \vdash_{\mathsf{m}} M' : D$. Therefore
$$\begin{aligned}\{[N]_{\vdash A}/x\}\,[\lambda\, y{:}C.\, M'] &\equiv \lambda\, y{:}\{[N]_{\vdash A}/x\}\,[\![C]\!]\,.\,\{[N]_{\vdash A}/x\}\,[M']\\ &\equiv \lambda\, y{:}[\![\{N/x\}C]\!]\,.\,[\{N/x\}M']_{\vdash \{N/x\}D}\\ &\equiv [\{N/x\}\,(\lambda\, y{:}C.\, M')]_{\vdash \{N/x\}(\Pi\, y C.\, D)}\end{aligned}$$
- Case $M'\, N'$. Then $B \equiv \{N'/y\}D$ where $\Gamma, x : A, \Gamma' \vdash_{\mathsf{m}} M' : \Pi\, y{:}C.\, D$ and $\Gamma, x : A, \Gamma' \vdash_{\mathsf{m}} N' : C$. Therefore
$$\begin{aligned}\{[N]_{\vdash A}/x\}\,[M\, N] &\equiv \{[N]_{\vdash A}/x\}\,[M']\,\{[N]_{\vdash A}/x\}\,[N']_{\vdash C}\\ &\equiv [\{N/x\}M']_{\vdash \{N/x\}(\Pi\, y C.\, D)}\,[\{N/x\}N']_{\vdash \{N/x\}C}\\ &\equiv [\{N/x\}\,M'\,N']_{\vdash \{N/x\}\{N'/y\}D}\end{aligned}$$

◀

Equivalence preservation

Having proved substitution preservation, we prove that the translation preserves equivalence: if two well-typed terms are equivalent in $\mathrm{CC}^{\infty}_{\subset}$ then their translations are equivalent in $\mathrm{CC}^{\infty}_{\uparrow}$.

▶ **Lemma 4.4** (Equivalence preservation). *If $\Gamma \vdash_{\subset} M : B$ and $\Gamma \vdash_{\subset} N : B$ and $M \equiv N$ then $[M]_{\Gamma \vdash B} \equiv [N]_{\Gamma \vdash B}$. If $\Gamma \vdash_{\subset} A : s$ and $\Gamma \vdash_{\subset} B : s$ and $A \equiv B$ then $[\![A]\!] \equiv [\![B]\!]$.*

Proof. By induction on the derivation of $M \equiv N$. The second statement derives from the first. We show the base case $(\lambda\, x{:}C.\, M')\, N' \equiv \{N'/x\}M'$. Then $B \equiv \{N'/y\}D$ where $\Gamma \vdash_{\subset} \lambda\, x{:}C.\, M' : \Pi\, x{:}C.\, D$ and $\Gamma \vdash_{\subset} N' : C$. Therefore

$$\begin{aligned}[(\lambda\, x{:}C.\, M')\, N']_{\vdash \{N'/x\}D} &\equiv \left(\lambda\, x{:}[C]_{\vdash s_1}\,.\,[M']_{\vdash D}\right)\,[N']_{\vdash C}\\ &\quad \text{using Proposition 4.2}\\ &\equiv \{[N']_{\vdash C}/x\}\,[M']_{\vdash D}\\ &\quad \text{by } \beta\text{-equivalence}\\ &\equiv [\{N'/x\}M']_{\vdash \{N'/x\}D}\\ &\quad \text{using Lemma 4.3}\end{aligned}$$

◀

Typing preservation

With substitution preservation and equivalence preservation at hand, we can finally prove the main theorem, namely that the translation preserves typing.

▶ **Lemma 4.5.** *The translation satisfies the following properties:*

- *For all* $s \in \mathcal{S}$, $\Gamma \vdash_\uparrow \mathsf{u}_s : [\![\mathcal{A}(s)]\!]$.
- *If* $\Gamma \vdash_\uparrow [A]_{\vdash s} : [\![s]\!]$ *then*

  $$\Gamma \vdash_\uparrow [A]_{\vdash \mathcal{N}(s)} : [\![\mathcal{N}(s)]\!] .$$

- *If* $\Gamma \vdash_\uparrow [A]_{\vdash s_1} : [\![s_1]\!]$ *and* $\Gamma, x : [\![A]\!] \vdash_\uparrow [B]_{\vdash s_2} : [\![s_2]\!]$ *then*

  $$\Gamma \vdash_\uparrow [\Pi\, x{:}A.\, B]_{\vdash \mathcal{R}(s_1,s_2)} : [\![\mathcal{R}(s_1, s_2)]\!] .$$

- *If* $\Gamma \vdash_\uparrow \mathsf{type}\,([\![A]\!])$ *and* $\Gamma, x : [\![A]\!] \vdash_\uparrow [M]_{\vdash B} : [\![B]\!]$ *then*

  $$\Gamma \vdash_\uparrow [\lambda\, x{:}A.\, M]_{\vdash \Pi\, xA.\, B} : [\![\Pi\, x{:}A.\, B]\!] .$$

- *If* $\Gamma \vdash_\uparrow [M]_{\vdash \Pi\, xA.\, B} : \Pi\, x{:}[\![A]\!].\, [\![B]\!]$ *and* $\Gamma \vdash_\uparrow [N]_{\vdash A} : [\![A]\!]$ *then*

  $$\Gamma \vdash_\uparrow [M\, N]_{\vdash \{N/x\}B} : [\![\{N/x\}B]\!] .$$

- *If* $\Gamma \vdash_\uparrow [M]_{\vdash A} : [\![A]\!]$ *and* $\Gamma \vdash_\uparrow \mathsf{type}\,(B)$ *and* $[\![A]\!] \equiv [\![B]\!]$ *then* $[\![\Gamma]\!] \vdash_\uparrow [M]_{\vdash B} : [\![B]\!]$.

Proof. Using Lemmas 4.2, 4.3, and 4.4. ◀

▶ **Theorem 4.6** (Typing preservation)**.** *If* $\Gamma \vdash_\subset M : A$ *then* $[\![\Gamma]\!] \vdash_\uparrow [M]_{\Gamma \vdash A} : [\![A]\!]_\Gamma$. *If* $\Gamma \vdash_\subset A : s$ *then* $[\![\Gamma]\!] \vdash_\uparrow \mathsf{type}\,([\![A]\!])$.

Proof. By induction on the derivation of $\Gamma \vdash_\subset M : A$, using Lemma 4.5. The second statement derives from the first.

- Case *variable*. Then $(x : [\![A]\!]) \in [\![\Gamma]\!]$ so $[\![\Gamma]\!] \vdash_\uparrow x : [\![A]\!]$.
- Case *sort*. By Lemma 4.5, $[\![\Gamma]\!] \vdash_\uparrow \mathsf{u}_s : [\![\mathcal{A}(s)]\!]$.
- Case *cumulativity*. By induction hypothesis, $[\![\Gamma]\!] \vdash_\uparrow [A]_{\vdash s} : [\![s]\!]$. By Lemma 4.5, $[\![\Gamma]\!] \vdash_\uparrow [A]_{\vdash \mathcal{N}(s)} : [\![\mathcal{N}(s)]\!]$.
- Case *product*. By induction hypothesis, $[\![\Gamma]\!] \vdash_\uparrow [A]_{\vdash s_1} : [\![s_1]\!]$ and $[\![\Gamma]\!], x : [\![A]\!] \vdash_\uparrow [B]_{\vdash s_2} : [\![s_2]\!]$. By Lemma 4.5,

  $$[\![\Gamma]\!] \vdash_\uparrow [\Pi\, x{:}A.\, B]_{\vdash \mathcal{R}(s_1,s_2)} : [\![\mathcal{R}(s_1, s_2)]\!] .$$

- Case *abstraction*. By induction hypothesis, $[\![\Gamma]\!] \vdash_\uparrow \mathsf{type}\,([\![A]\!])$ and $[\![\Gamma]\!], x : [\![A]\!] \vdash_\uparrow [M]_{\vdash B} : [\![B]\!]$. By Lemma 4.5,

  $$\Gamma \vdash_\uparrow [\lambda\, x{:}A.\, M]_{\vdash \Pi\, xA.\, B} : [\![\Pi\, x{:}A.\, B]\!] .$$

- Case *application*. By induction hypothesis, $[\![\Gamma]\!] \vdash_\uparrow [M]_{\vdash \Pi\, xA.\, B} : \Pi\, x : [\![A]\!].\, [\![B]\!]$ and $[\![\Gamma]\!] \vdash_\uparrow [N]_{\vdash A} : [\![A]\!]$. By Lemma 4.5,

  $$[\![\Gamma]\!] \vdash_\uparrow [M\, N]_{\vdash \{N/x\}B} : [\![\{N/x\}B]\!] .$$

- Case *conversion*. By induction hypothesis, $[\![\Gamma]\!] \vdash_\uparrow [M]_{\vdash A} : [\![A]\!]$ and $[\![\Gamma]\!] \vdash_\uparrow \mathsf{type}\,([\![B]\!])$. By Lemma 4.5, $[\![\Gamma]\!] \vdash_\uparrow [M]_{\vdash B} : [\![B]\!]$.

◀

▶ **Corollary 4.7.** *If* $\mathsf{WF}_\subset(\Gamma)$ *then* $\mathsf{WF}_\uparrow([\![\Gamma]\!])$.

Proof. By induction on Γ. ◀

5 Operational semantics

We presented $\mathrm{CC}^\infty_\uparrow$ assuming the equivalence relation $\equiv$ satisfies the equations of Definition 3.2. In practice, such equivalence relations are defined as the congruence closure of a set of reduction rules. In the case of $\mathrm{CC}^\infty_\subset$, it is the closure of β-reduction $\longrightarrow_\beta$, which enjoys confluence, subject reduction, and strong normalization. We now do the same for $\mathrm{CC}^\infty_\uparrow$.

Rewrite rules

The equations for the decoding function $\mathsf{T}_s\,(A)$ are easily oriented into rewrite rules:

$$\begin{aligned}
\mathsf{T}_{\mathcal{A}(s)}\,(\mathsf{u}_s) &\longrightarrow \mathsf{U}_s \\
\mathsf{T}_{\mathcal{N}(s)}\,(\uparrow_s\,(A)) &\longrightarrow \mathsf{T}_s\,(A) \\
\mathsf{T}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\,x : A.B) &\longrightarrow \Pi\,x{:}\mathsf{T}_{s_1}\,(A)\,.\,\mathsf{T}_{s_2}\,(B)\,.
\end{aligned}$$

With these rules, we can view $\mathsf{T}_s\,(A)$ as a recursively defined function that decodes terms of type U_s into types by traversing their structure.

Orienting the equations for $\uparrow_s$ is more delicate. In Martin-Löf's intuitionistic type theory, a single equation is needed to reflect equality:

$$\uparrow_i\,(\pi_{i,i}\,x : A.B) \equiv \pi_{i+1,i+1}\,x : \uparrow_i\,(A)\,.\uparrow_i\,(B)\,.$$

In that case, it seems natural to orient the equation from left to right and see $\uparrow_i$ as a function that recursively transforms codes in U_i into equivalent codes in U_{i+1}:

$$\uparrow_i\,(\pi_{i,i}\,x : A.B) \longrightarrow \pi_{i+1,i+1}\,x : \uparrow_i\,(A)\,.\uparrow_i\,(B)\,.$$

While elegant, that solution does not behave well with the impredicative universe Prop. The equation

$$\pi_{i+1,\mathsf{Prop}}\,x : \uparrow_i\,(A)\,.B \equiv \pi_{i,\mathsf{Prop}}\,x : A.B$$

requires the rewrite rule

$$\pi_{i+1,\mathsf{Prop}}\,x : \uparrow_i\,(A)\,.B \longrightarrow \pi_{i,\mathsf{Prop}}\,x : A.B$$

which would break confluence with the previous rule because of the critical pair

$$\pi_{i+1,\mathsf{Prop}}\,x : \uparrow_i\,(\pi_{i,i}\,y : A.B)\,.C.$$

Fortunately, we can still orient the equations in the other direction and obtain a well-behaved system. Again, we can express this concisely using the $\uparrow^{s_2}_{s_1}\,(A)$ notation.

▶ **Definition 5.1.** The equivalence relation $\equiv$ in $\mathrm{CC}^\infty_\uparrow$ is defined as the congruence induced by the following set of rewrite rules:

$$\begin{aligned}
(\lambda\,x{:}A.\,M)\;N &\longrightarrow_\beta \{N/x\}M \\
\mathsf{T}_{\mathcal{A}(s)}\,(\mathsf{u}_s) &\longrightarrow_\tau \mathsf{U}_s \\
\mathsf{T}_{\mathcal{N}(s)}\,(\uparrow_s\,(A)) &\longrightarrow_\tau \mathsf{T}_s\,(A) \\
\mathsf{T}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\,x : A.B) &\longrightarrow_\tau \Pi\,x{:}\mathsf{T}_{s_1}\,(A)\,.\,\mathsf{T}_{s_2}\,(B) \\
\pi_{\mathcal{N}(s_1),s_2}\,x : \uparrow_{s_1}\,(A)\,.B &\longrightarrow_\sigma \uparrow^{\mathcal{R}(\mathcal{N}(s_1),s_2)}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\,x : A.B) \\
\pi_{s_1,\mathcal{N}(s_2)}\,x : A.\uparrow_{s_2}\,(B) &\longrightarrow_\sigma \uparrow^{\mathcal{R}(s_1,\mathcal{N}(s_2))}_{\mathcal{R}(s_1,s_2)}\,(\pi_{s_1,s_2}\,x : A.B)\,.
\end{aligned}$$

In this formulation, the coercions $\uparrow_s$ propagate upwards towards the root of the term. This behavior matches the idea that, when computing minimal types, the cumulativity rule should be delayed as much as possible.

Properties

We show that the rewrite system $\longrightarrow_{\beta\tau\sigma}$ enjoys the usual properties of confluence, subject-reduction, and strong normalization. The last one follows from the strong normalization of $\mathrm{CC}^{\infty}_{\subset}$.

▶ **Theorem 5.2** (Normalization of $\longrightarrow_{\tau\sigma}$). *The rewrite system $\longrightarrow_{\tau\sigma}$ is terminating.*

Proof. The relation $\longrightarrow_{\tau}$ strictly decreases the total height of T_s symbols and the relation $\longrightarrow_{\sigma}$ strictly decreases the total depth of $\uparrow_s$ symbols (while leaving the height of T_s unchanged), therefore $\longrightarrow_{\tau\sigma}$ is terminating. ◀

▶ **Theorem 5.3** (Confluence). *The rewrite system $\longrightarrow_{\beta\tau\sigma}$ is locally confluent.*

Proof. The rewrite rules of $\longrightarrow_{\tau\sigma}$ are left-linear and the critical pairs are convergent, therefore $\longrightarrow_{\tau\sigma}$ is locally confluent. By Proposition 5.2, it is terminating and hence confluent. Therefore its union with $\longrightarrow_{\beta}$ is confluent [23]. ◀

▶ **Theorem 5.4** (Subject reduction). *If $\Gamma \vdash_{\uparrow} M : A$ and $M \longrightarrow_{\beta\tau\sigma} M'$ then $\Gamma \vdash_{\uparrow} M' : A$.*

Proof. By induction on M. ◀

▶ **Theorem 5.5** (Strong normalization). *The rewrite system $\longrightarrow_{\beta\tau\sigma}$ is strongly normalizing for well-typed terms.*

Proof. By Theorem 5.2, $\longrightarrow_{\tau\sigma}$ is terminating, so any infinite sequence of reductions must have an infinite number of $\longrightarrow_{\beta}$ steps. If $M \longrightarrow_{\tau\sigma} M'$ then $|M| = |M'|$. If $M \longrightarrow_{\beta} M'$ then $|M| \longrightarrow_{\beta} |M'|$. An infinite reduction sequence in $\mathrm{CC}^{\infty}_{\uparrow}$ would therefore lead to an infinite reduction sequence in $\mathrm{CC}^{\infty}_{\subset}$. Moreover, according to Theorem 3.7 and Proposition 5.4, the sequence would be well-typed. Since $\mathrm{CC}^{\infty}_{\subset}$ is strongly normalizing [11], this is impossible. ◀

6 Conclusion

We presented a formulation of the cumulative calculus of constructions with explicit subtyping. We used the Tarski style of universes to solve the issues related to dependent types and coercions. We showed that, by reflecting equality, we were able to preserve the expressiveness of Russell-style universes.

A thorough and definitive study of the two styles remains to be done. Are the two styles always equivalent? Can we always define an equivalence relation that reflects equality? Can it always be oriented into well-behaved rewrite rules? Finally, how does this solution interact with product covariance or other extensions of the theory, such as inductive types or universe polymorphism? Our guess is that inductive types should not pose a problem. Product covariance could be handled either by pre-expanding the terms to η-long form or by using a more general form of coercions $\uparrow^B_A$ where $A \leq B$. The interaction with universe polymorphism is still unclear.

Our results connect work done in pure type systems to work done in Martin-Löf's intuitionistic type theory. While the two theories have a clearly related core (namely the λ-calculus with dependent types), it is less obvious if they can still be unified or if they have definitively diverged. Pure type systems allow for a wide variety of specifications while intuitionistic type theory has a clear and intuitive interpretation for cumulativity. We feel that this problem deserves to be studied as the two theories form the basis for many logical

frameworks and proof assistants. The work of Herbelin and Siles [19], and van Doorn et al [22] already showed some progress in this direction.

A requirement for the aforementioned program is the development of a notion of cumulativity in pure type systems. We can imagine extending PTS specifications with a cumulativity relation as done by Barras, Grégoire, and Lasson for example [2, 3, 10]. However, it is unclear if such an extension is meaningful on its own, or if it only makes sense in $\mathrm{CC}^{\infty}_{\subset}$ (which is both a functional *and* complete PTS). In particular, the equations of $\mathrm{CC}^{\infty}_{\uparrow}$ rely on the fact that lifting inside a product cannot decrease the type of the product: $\mathcal{R}(s_1, s_2) \subseteq \mathcal{R}(\mathcal{N}(s_1), s_2)$ and $\mathcal{R}(s_1, s_2) \subseteq \mathcal{R}(s_1, \mathcal{N}(s_2))$. Whether this condition is essential or whether it can be avoided is unclear. The possibility of using universes à la Tarski remains to be studied.

Finally, while our system allowed us to get rid of the implicit subsumption rule, it did so at the expense of some complexity in the conversion rule. Whether this trade-off is beneficial in practical applications remains to be discussed. How does the Tarski style simplify the theoretical studies of the calculus of constructions? Can the current implementation of the calculus of constructions like Coq or Matita benefit from it? Nevertheless, this presentation is better suited for logical frameworks such as Dedukti, which usually do not support subtyping as a built-in. Our work opens the way for exporting Coq proofs to such frameworks.

Acknowledgments. We thank Gilles Dowek and Raphaël Cauderlier for the discussions leading to the ideas behind this paper and their feedback throughout its lengthy writing process, as well as the anonymous reviewers for their various suggestions on how to improve it.

References

1 Henk Barendregt. Lambda calculi with types. In Samson Abramsky, Dov M. Gabbay, and Thomas S. E. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 117–309. Oxford University Press, 1992.

2 Bruno Barras. *Auto-validation d'un système de preuves avec familles inductives*. PhD thesis, Université Paris 7, 1999.

3 Bruno Barras and Benjamin Grégoire. On the role of type decorations in the calculus of inductive constructions. In Luke Ong, editor, *Computer Science Logic*, number 3634 in Lecture Notes in Computer Science, pages 151–166. Springer Berlin Heidelberg, 2005.

4 Mathieu Boespflug and Guillaume Burel. CoqInE: Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *Proof Exchange for Theorem Proving–Second International Workshop, PxTP*, page 44, 2012.

5 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-Pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *Typed Lambda Calculi and Applications*, number 4583 in Lecture Notes in Computer Science, pages 102–117. Springer Berlin Heidelberg, 2007.

6 Herman Geuvers and Freek Wiedijk. A logical framework with explicit conversions. *Electronic Notes in Theoretical Computer Science*, 199:33–47, February 2008.

7 Robert Harper and Robert Pollack. Type checking, universe polymorphism, and typical ambiguity in the calculus of constructions draft. In J. Díaz and F. Orejas, editors, *TAPSOFT'89*, number 352 in Lecture Notes in Computer Science, pages 241–256. Springer Berlin Heidelberg, 1989.

8 Robert Harper and Robert Pollack. Type checking with universes. *Theoretical Computer Science*, 89(1):107–136, October 1991.

9 Hugo Herbelin and Arnaud Spiwack. The Rooster and the Syntactic Bracket. 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 169–187, Dagstuhl, Germany, 2014. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

10 Marc Lasson. *Réalisabilité et paramétricité dans les systèmes de types purs*. PhD thesis, Ecole normale supérieure de Lyon, 2012.

11 Zhaohui Luo. $CC^{\infty}_{\subset}$ and its meta theory. *Laboratory for Foundations of Computer Science Report ECS-LFCS-88-58*, 1988.

12 Zhaohui Luo. ECC, an extended calculus of constructions. In *Fourth Annual Symposium on Logic in Computer Science, 1989. LICS'89, Proceedings*, pages 386–395, June 1989.

13 Zhaohui Luo. *Computation and Reasoning: A Type Theory for Computer Science*. Oxford University Press, Inc., New York, NY, USA, 1994.

14 Zhaohui Luo. Notes on universes in type theory. Lecture notes for a talk at Institute for Advanced Study, Princeton (`http://www.cs.rhul.ac.uk/home/zhaohui/universes.pdf`), 2012.

15 Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, 1984.

16 Erik Palmgren. On universes in type theory. In *Twenty-five years of constructive type theory*, pages 191–204. Oxford University Press, October 1998.

17 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

18 Ronan Saillard. Dedukti: a universal proof checker. In *Foundation of Mathematics for Computer-Aided Formalization Workshop*, 2013.

19 Vincent Siles and Hugo Herbelin. Pure type system conversion is always typable. *Journal of Functional Programming*, 22(02):153–180, 2012.

20 The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

21 L. S. van Benthem Jutting, J. McKinna, and R. Pollack. Checking algorithms for pure type systems. In Henk Barendregt and Tobias Nipkow, editors, *Types for Proofs and Programs*, number 806 in Lecture Notes in Computer Science, pages 19–61. Springer Berlin Heidelberg, 1994.

22 Floris van Doorn, Herman Geuvers, and Freek Wiedijk. Explicit convertibility proofs in pure type systems. In *Proceedings of the Eighth ACM SIGPLAN International Workshop on Logical Frameworks & Meta-languages: Theory & Practice*, LFMTP'13, pages 25–36, New York, NY, USA, 2013. ACM.

23 Vincent van Oostrom. *Confluence for Abstract and Higher-Order Rewriting*. PhD thesis, Vrije Universiteit, Amsterdam, 1994.

Objects and Subtyping in the λΠ-Calculus Modulo

Raphaël Cauderlier[1,2] and Catherine Dubois[2,3]

1 **INRIA Paris-Rocquencourt, Paris, France**
2 **CNAM, Paris, France**
3 **ENSIIE, Évry, France**

Abstract

We present a shallow embedding of the Object Calculus of Abadi and Cardelli in the λΠ-calculus modulo, an extension of the λΠ-calculus with rewriting. This embedding may be used as an example of translation of subtyping. We prove this embedding correct with respect to the operational semantics and the type system of the Object Calculus. We implemented a translation tool from the Object Calculus to Dedukti, a type-checker for the λΠ-calculus modulo.

1998 ACM Subject Classification F.4.1 Lambda calculus and related systems

Keywords and phrases object, calculus, encoding, dependent type, rewrite system

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.47

1 Introduction

Motivation. The λΠ-calculus modulo [16] (λΠm) is a type system with dependent types in which the conversion congruence can be extended by a user-supplied rewrite system. It can be used as a logical framework to encode all the functional Pure Type Systems [16]. Moreover, translation tools from real-world proof assistants like Coq [15, 4] and the HOL family [3] to Dedukti [6], a type-checker for λΠm, allow for the verification of proofs done in these complex systems using a small, easy to trust, checker.

In this paper we present an encoding of an object calculus in λΠm, more precisely the simply-typed ς-calculus [2]. A major feature of object oriented type systems is subtyping, and it will be the focus of this article. The simply-typed ς-calculus is the simplest object calculus featuring subtyping. We chose it as our source language to understand the special case of structural object subtyping to be compared with other forms of subtyping like universe cumulativity in Coq or predicate subtyping in PVS.

We also believe that objects may be useful for proof assistants like they already are for programming; we would like to be able to develop proofs using object oriented concepts and mechanisms such as inheritance, method redefinition and late binding. FoCaLiZe [24] is a logical system featuring class-based object mechanisms which are translated in λΠm [9]. In order to generalize this encoding of objects in λΠm to more primitive object-based mechanisms, we would need complex objects where methods would be typed with dependent types. This work is a first step in that direction starting from a very simple type-system for objects.

Related work. Many encodings [31, 7] of objects have been developed, studied, and compared in the 90s. In order to express complex but common object mechanisms such as self reference and inheritance, the target language is usually chosen to be very rich like System $F^{\omega}_{<:}$ (a type system featuring polymorphism, existential types, type operators and subtyping).

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 47–71

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

Because of the complexity of System $F^{\omega}_{<:}$ and the limitations of these encodings, they are of limited practicality to study object oriented languages or to implement object oriented mechanisms in proof systems; the only implementation to our knowledge is the Yarrow proof assistant [32].

However, following the example of the λ-calculus, small calculi taking objects as core notions have been designed and their type systems have been proved safe. For example:

- The λ-calculus of objects [18] is an extension of the simply-typed λ-calculus with object construction, method call, and method update. In this system, objects are extended with their types using extensible records.
- The Object Calculi of Abadi and Cardelli [2] are a collection of calculi based on objects. They differ from the λ-calculus of objects in two important ways: they are not based on the λ-calculus so they have fewer constructs and objects and their types are fixed records. Hence they are somewhat simpler but they still are very expressive.
- Featherweight Java [25] is a core calculus for the popular class-based Java programming language. It is a small class-based object oriented calculus designed to study extensions of class-based languages such as Java.

These three calculi can easily encode the λ-calculus, allow possibly non-terminating recursion and have some form of subtyping: respectively row polymorphism, structural subtyping, and class-based subtyping.

The type systems of these three calculi have been formalized in proof assistants; those formalizations can be seen as deep embeddings of the calculi in type theory. For example, Featherweight Java has been formalized in Coq [26] and Isabelle/HOL [19] using extensible records and subject reduction for Object Calculi has been proved in Coq [11] and Isabelle/HOL [23]. For the untyped Object Calculus, confluence has also been formally proved in Isabelle/HOL [22].

Encodings of objects based on rewrite techniques have also been studied; for example, in the ρ-calculus [13], a full encoding of the untyped Object Calculus and λ-calculus of objects [12] and a partial encoding of the simply-typed Object Calculus [13] have been designed. In the Maude specification environment [14], objects are also encoded using a rewrite system thanks to the reflection mechanism of Maude.

Contribution. In contrast with these deep encodings, our contribution is a shallow embedding in the sense of [8, 5, 17]; the elements of the source language, the simply-typed ς-calculus: terms, values, and types are respectively translated to terms, values, and types in λΠm such that operational semantics, typing derivations, and binding operation are preserved by this translation.

The next section of this article describes λΠm, our target language, Section 3 describes the simply-typed ς-calculus, our source language. Section 4 is the main section of this article; it defines a strongly-normalizing encoding of the simply-typed ς-calculus in λΠm. This encoding is not fully shallow because it does not preserve the operational semantics. In Section 5, we add two rewrite rules to this encoding to reflect the operational semantics; doing so we lose strong-normalization.

2 The λΠ-calculus modulo

2.1 The λΠ-calculus

The λΠ-calculus [20], also known as LF and λP, is an extension of the simply typed λ-calculus with dependent types. λΠ terms and types have the following syntax:

$$s \in \{\texttt{Type}, \texttt{Kind}\}$$

$$\frac{}{\emptyset \vdash_d} \text{(Empty)} \qquad \frac{\Gamma \vdash_d \qquad \Gamma \vdash_d \tau : s \qquad x \notin \Gamma}{\Gamma, x : \tau \vdash_d} \text{(Decl)} \qquad \frac{\Gamma \vdash_d}{\Gamma \vdash_d \texttt{Type} : \texttt{Kind}} \text{(Sort)}$$

$$\frac{\Gamma \vdash_d \qquad x : \tau \in \Gamma}{\Gamma \vdash_d x : \tau} \text{(Var)} \qquad \frac{\Gamma \vdash_d \tau_1 : \texttt{Type} \qquad \Gamma, x : \tau_1 \vdash_d \tau_2 : s}{\Gamma \vdash_d \Pi x : \tau_1 . \tau_2 : s} \text{(Prod)}$$

$$\frac{\Gamma \vdash_d \tau_1 : \texttt{Type} \qquad \Gamma, x : \tau_1 \vdash_d \tau_2 : s \qquad \Gamma, x : \tau_1 \vdash_d t : \tau_2}{\Gamma \vdash_d \lambda x : \tau_1 . t : \Pi x : \tau_1 . \tau_2} \text{(Abs)}$$

$$\frac{\Gamma \vdash_d t_0 : \Pi x : \tau_1 . \tau_2 \qquad \Gamma \vdash_d t_1 : \tau_1}{\Gamma \vdash_d t_0 t_1 : \tau_2\{t_1/x\}} \text{(App)}$$

$$\frac{\Gamma \vdash_d t : \tau_1 \qquad \Gamma \vdash_d \tau_1 : s \qquad \Gamma \vdash_d \tau_2 : s \qquad \tau_1 \equiv_\beta \tau_2}{\Gamma \vdash_d t : \tau_2} \text{(Conv)}$$

Figure 1 inference rules for the $\lambda\Pi$-calculus.

$$t, u, v, \ldots, \tau ::= x \mid t\ u \mid \lambda x : \tau . t \mid \Pi x : \tau_1 . \tau_2 \mid \texttt{Type} \mid \texttt{Kind}$$

There is no syntactic distinction between terms and types but we use latin letters starting at t to denote terms and the greek letter τ to denote types. We use the letter s to denote a sort, either `Type` or `Kind`. The term $\Pi x : \tau_1 . \tau_2$ where the variable x may appear free in τ_2 is called a dependent product and represents the type of functions taking an argument x of type τ_1 and returning a value of type τ_2 that may depend on x. If x does not appear free in τ_2, the term $\Pi x : \tau_1 . \tau_2$ will be abbreviated as $\tau_1 \to \tau_2$. If τ_1 is clear from context, the term $\Pi x : \tau_1 . \tau_2$ will be abbreviated as $\Pi x . \tau_2$.

A list of variable typing declarations is called a ($\lambda\Pi$) context:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau$$

where $\emptyset$ denotes the empty context. We implicitly use α-conversion to avoid variable capture. In particular, contexts contain distinct variables.

Some contexts are called well-formed. When the context Γ is well-formed, we write $\Gamma \vdash_d$. Some terms are called well-typed. When the term t is well-typed of type τ in context Γ, we write $\Gamma \vdash_d t : \tau$. These two notions are mutually defined in Figure 1 where $t_0\{t_1/x\}$ denotes the capture-avoiding substitution of the variable x by the term t_1 in term t_0 and $\equiv_\beta$ is the congruence induced by β-reduction (the smallest congruence such that $(\lambda x : \tau_1 . t_0) t_1 \equiv_\beta t_0\{t_1/x\}$).

The $\lambda\Pi$-calculus is the type-system on which logical frameworks such as Automath [28] and Twelf [30] are based.

2.2 The $\lambda\Pi$-calculus modulo

The $\lambda\Pi$-calculus modulo ($\lambda\Pi$m) is an extension of the $\lambda\Pi$-calculus which extends the conversion rule; terms are considered convertible not only when they are β-equivalent but also when they are congruent for a given rewrite system.

The terms are the same as in the $\lambda\Pi$-calculus but contexts may also contain rewrite rules which also need to be well-typed.

Rewrite rules are composed of three parts: a rule context which is a λΠ context used to type free variables, a left-hand side and a right-hand side which are both terms. In order to make the rewrite system decidable[1], we need to add the following restrictions on rewrite rules:

- the left-hand side is a first-order pattern (a term built only from variables and applications)
- free variables of the right-hand side also appear free in the left-hand side
- free variables of the left-hand side are declared in the rule context.

So the new syntax for contexts is as follows:

$$\Gamma ::= \emptyset \mid \Gamma, x : \tau \mid \Gamma, (\Lambda t \hookrightarrow u)$$

where Λ stands for λΠ contexts. The rule context Λ will often be omitted when clear from context.

For any context Γ, a reduction relation on terms $\longrightarrow_{\beta\Gamma}$ is defined by:

- for any terms t_1, t_2 and any variable x, $(\lambda x.t_1)t_2 \longrightarrow_{\beta\Gamma} t_1\{t_2/x\}$
- for any rule $(\Lambda l \hookrightarrow r) \in \Gamma$ and any substitution θ of the variables of Λ, $\theta l \longrightarrow_{\beta\Gamma} \theta r$.

We denote by $\equiv_{\beta\Gamma}$ the smallest congruence containing $\longrightarrow_{\beta\Gamma}$.

To check if contexts are well-formed, we add a rule for the new case of rewrite rule. A rewrite rule is well-formed in a context Γ if the left-hand side and the right-hand side have the same type in Γ, Λ (Γ augmented with the rule context):

$$\frac{\Gamma \vdash_d \qquad \Gamma, \Lambda \vdash_d t : \tau \qquad \Gamma, \Lambda \vdash_d u : \tau}{\Gamma, (\Lambda t \hookrightarrow u) \vdash_d} \text{(RewriteRule)}$$

The set of rewrite rules in a context Γ defines a rewrite system; the conversion rule for λΠm is the same as the one for the λΠ-calculus except that the β-equivalence is replaced by the congruence $\equiv_{\beta\Gamma}$.

$$\frac{\Gamma \vdash_d t : \tau_1 \qquad \Gamma \vdash_d \tau_1 : s \qquad \Gamma \vdash_d \tau_2 : s \qquad \tau_1 \equiv_{\beta\Gamma} \tau_2}{\Gamma \vdash_d t : \tau_2} \text{(Conv)}$$

Other typing rules are unchanged. In particular, if the typing judgment $\Gamma \vdash_d t : T$ is derivable in the λΠ-calculus, then it is also derivable in λΠm with the exact same derivation and an empty rewrite system.

An example of well-formed λΠm-context[2] is shown in Figure 2. This example is composed of the definitions of the addition in Peano arithmetic and the concatenation of lists depending on their length. Here and in rest of the paper, we omit in such definitions the types of variables introduced by Π and λ when it is not ambiguous. The definition of the addition is needed to convert the types of the left-hand side to the type of the right-hand side of each rewrite rule defining the concatenation; for instance, let us check that the rule $\texttt{append}\ 0\ n\ \texttt{empty}\ l \hookrightarrow l$ is well-formed in the context $\Gamma := \texttt{Nat} : \texttt{Type}, 0 : \texttt{Nat}, \ldots, \texttt{append} : \Pi n_1.\Pi n_2.\texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1\ n_2)$:

[1] That is, to decide whether a given term matches a rewrite rule.

[2] Examples and other contexts in λΠm are preceded in this article by a vertical bar in order to distinguish them from examples in the ς-calculus.

$$
\begin{array}{ll}
 & \texttt{A} : \texttt{Type}. \\
\texttt{Nat} : \texttt{Type}. & \texttt{List} : \texttt{Nat} \to \texttt{Type}. \\
0 : \texttt{Nat}. & \texttt{empty} : \texttt{List}\ 0. \\
\texttt{S} : \texttt{Nat} \to \texttt{Nat}. & \texttt{cons} : \Pi n : \texttt{Nat}.\texttt{A} \to \texttt{List}\ n \to \texttt{List}\ (\texttt{S}\ n). \\
 & \\
\texttt{plus} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Nat}. & \texttt{append} : \Pi n_1.\Pi n_2. \\
 & \quad \texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1 n_2). \\
 & \\
\texttt{plus}\ 0\ n \hookrightarrow n. & \texttt{append}\ 0\ n\ \texttt{empty}\ l \hookrightarrow l. \\
\texttt{plus}\ n\ 0 \hookrightarrow n. & \texttt{append}\ n\ 0\ l\ \texttt{empty} \hookrightarrow l. \\
\texttt{plus}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2). & \texttt{append}\ (\texttt{S}\ n_1)\ n_2\ (\texttt{cons}\ n_1\ a\ l_1)\ l_2 \hookrightarrow \\
\texttt{plus}\ n_1\ (\texttt{S}\ n_2) \hookrightarrow \texttt{S}\ (\texttt{plus}\ n_1\ n_2). & \quad \texttt{cons}\ (\texttt{plus}\ n_1\ n_2)\ a\ (\texttt{append}\ n_1\ n_2\ l_1\ l_2)
\end{array}
$$

Figure 2 Example of λΠm-context: Peano natural numbers and concatenation of dependent lists.

$$
\begin{array}{l}
\texttt{equal} : \texttt{Nat} \to \texttt{Nat} \to \texttt{Type}. \\
\texttt{refl} : \Pi n : \texttt{Nat}.\texttt{equal}\ n\ n. \\
 \\
\texttt{equal\_S} : \Pi n_1.\Pi n_2.\texttt{equal}\ n_1\ n_2 \to \texttt{equal}\ (\texttt{S}\ n_1)\ (\texttt{S}\ n_2). \\
\texttt{equal\_S}\ n\ n\ (\texttt{refl}\ \texttt{n}) \hookrightarrow \texttt{refl}\ (\texttt{S}\ n). \\
 \\
\texttt{plus\_comm} : \Pi n_1.\Pi n_2.\texttt{equal}\ (\texttt{plus}\ n_1\ n_2)\ (\texttt{plus}\ n_2\ n_1). \\
\texttt{plus\_comm}\ 0\ n_2 \hookrightarrow \texttt{refl}\ n_2. \\
\texttt{plus\_comm}\ (\texttt{S}\ n_1)\ n_2 \hookrightarrow \texttt{equal\_S}\ (\texttt{plus}\ n_1\ n_2)\ (\texttt{plus}\ n_2\ n_1)\ (\texttt{plus\_comm}\ n_1\ n_2).
\end{array}
$$

Figure 3 A proof of the commutativity of addition in λΠm.

- The implicit rule context is $\Lambda := (n : \texttt{Nat}, l : \texttt{List}\ n)$.
- The constants 0, `empty`, and `append` have respectively the types `Nat`, `List` 0, and $\Pi n_1.\Pi n_2.\texttt{List}\ n_1 \to \texttt{List}\ n_2 \to \texttt{List}\ (\texttt{plus}\ n_1\ n_2)$ in Γ.
- By successive applications of the (App) rule, we can type the left-hand side $\texttt{append}\ 0\ n\ \texttt{empty}\ l$ with type $\texttt{List}\ (\texttt{plus}\ 0\ n)$ in Γ, Λ.
- The rule $\texttt{plus}\ 0\ n \hookrightarrow n$ is in Γ so $\texttt{List}\ (\texttt{plus}\ 0\ n) \equiv_{\beta(\Gamma,\Delta)} \texttt{List}\ n$, therefore we can also type the left-hand side with the type $\texttt{List}\ n$ in context Γ, Λ using the (Conv) rule.
- The left-hand side $\texttt{append}\ 0\ n\ \texttt{empty}\ l$ and the right-hand side l have the same type $\texttt{List}\ n$ in context Γ, Λ hence the rule $\Lambda(\texttt{append}\ 0\ n\ \texttt{empty}\ l) \hookrightarrow l$ is well-formed in context Γ.

Thanks to dependent types, we can state and prove theorems in λΠm. To state a theorem, we declare a symbol whose type is the theorem statement and to prove the theorem we add one or more rewrite rules defining this symbol as a (total and terminating) function. A λΠm proof of the addition commutativity is given in Figure 3. This proof is composed of two rewrite rules that mimic a proof by induction on the first argument of `plus`. In the following, we call such a proof scheme a λΠm induction proof.

The interesting properties about a λΠm-context and its associated rewrite system are confluence, strong normalization and well-formedness. None of them is decidable but when the rewrite system is both confluent and strongly normalizing, convertibility check can be

decided by comparing normal forms so well-formedness becomes decidable and is indeed implemented in the Dedukti [6] type checker.

However, the correctness of Dedukti relies on confluence only; strong normalization is only used to ensure termination.

3 The simply-typed ς-calculus

In this section, we describe the source language of our encoding, that is the simply-typed ς-calculus defined by Abadi and Cardelli [2, 1] (also called $Obj_{1<:}$). This calculus is an object-based (classes are not primitive constructs) calculus with functional semantics (values are immutable). Its type system features structural subtyping (as opposed to class subtyping). Contrary to simply-typed λ-calculus, well-typed ς-terms do not always terminate.

3.1 Syntax

The syntax of the simply-typed ς-calculus is divided between types and terms.

Types are (possibly empty) records of types:

$$A, B, \ldots ::= [l_i : A_i]_{i \in 1 \ldots n}$$

Labels are distinct and their order does not matter as long as each l_i remains associated to the same A_i. Terms are records of methods introduced by a self binder ς. Methods can be selected and updated.

$$\begin{array}{rcll} a, b, \ldots & ::= & x & \textit{variable} \\ & | & [l_i = \varsigma(x_i : A) a_i]_{i \in 1 \ldots n} & \textit{object} \\ & | & a.l & \textit{method selection} \\ & | & a.l \Leftarrow \varsigma(x : A) b & \textit{method update} \end{array}$$

Again, labels in objects are distinct and their order does not matter. When the variable introduced by the ς binder is unused, we may omit the binder and write $l = b$ and $a.l \Leftarrow b$ instead of, respectively, $l = \varsigma(x : A)b$ and $a.l \Leftarrow \varsigma(x : A)b$ where x does not appear free in b.

Typing contexts are lists of typing declarations:

$$\Delta ::= \emptyset \mid \Delta, x : A$$

in which each variable may appear at most once. When x appears in Δ, we denote by $\Delta(x)$ the associated type.

3.2 Typing

The following rules, where A stands for $[l_i : A_i]_{i \in 1 \ldots n}$, define a type system for the simply-typed ς-calculus[3]:

[3] Abadi and Cardelli also consider a ground type that they call K or Top to ease comparison with the simply-typed λ-calculus. It can be replaced by the empty object type [ ] so we omit it here to simplify the calculus.

$$\frac{\Delta \vdash_\varsigma A_i \qquad \forall i \in 1 \dots n \qquad l_i \text{ distinct}}{\Delta \vdash_\varsigma A} \text{(type)} \qquad \frac{}{\emptyset \vdash_\varsigma} \text{(empty)}$$

$$\frac{\Delta \vdash_\varsigma A \qquad x \notin \Delta}{\Delta, x : A \vdash_\varsigma} \text{(decl)} \qquad \frac{\Delta \vdash_\varsigma \qquad x \in \Delta}{\Delta \vdash_\varsigma x : \Delta(x)} \text{(var)}$$

$$\frac{\Delta, x_i : A \vdash_\varsigma a_i : A_i \qquad \forall i \in 1 \dots n}{\Delta \vdash_\varsigma [l_i = \varsigma(x_i : A)a_i]_{i \in 1 \dots n} : A} \text{(obj)} \qquad \frac{\Delta \vdash_\varsigma a : A \qquad j \in 1 \dots n}{\Delta \vdash_\varsigma a.l_j : A_j} \text{(select)}$$

$$\frac{\Delta \vdash_\varsigma a : A \qquad \Delta, x : A \vdash_\varsigma b : A_j \qquad j \in 1 \dots n}{\Delta \vdash_\varsigma a.l_j \Leftarrow \varsigma(x : A)b : A} \text{(update)}$$

3.2.1 Subtyping

This type system is extended by a subtyping relation $<:$ defined as follows:

$$\frac{\Delta \vdash_\varsigma A_i \qquad \forall i \in 1 \dots n+m}{\Delta \vdash_\varsigma [l_i : A_i]_{i \in 1 \dots n+m} <: [l_i : A_i]_{i \in 1 \dots n}} \text{(subtype)} \qquad \frac{\Delta \vdash_\varsigma A}{\Delta \vdash_\varsigma A <: A} \text{(refl)}$$

$$\frac{\Delta \vdash_\varsigma A <: B \qquad \Delta \vdash_\varsigma B <: C}{\Delta \vdash_\varsigma A <: C} \text{(trans)}$$

Since the order of labels is irrelevant, the (subtype) rule actually states that A is a subtype of B whenever every label of B is also in A, with the same type. This subtyping relation can be used to change the type of terms with the following subsumption rule:

$$\frac{\Delta \vdash_\varsigma a : A \qquad \Delta \vdash_\varsigma A <: B}{\Delta \vdash_\varsigma a : B} \text{(subsume)}$$

3.2.2 Minimum types

Abadi and Cardelli have proved that the simply-typed ς-calculus enjoys minimum typing [2]: for each well-typed term a in a context Δ, we can compute a type $\mathbf{mintype}_\Delta(a)$ such that:

- $\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta(a)$
- for all A such that $\Delta \vdash_\varsigma a : A$, we have $\Delta \vdash_\varsigma \mathbf{mintype}_\Delta(a) <: A$.

The meta-level function $\mathbf{mintype}$[4] is defined as follows:

$\mathbf{mintype}_\Delta(x) := \Delta(x)$ $\qquad$ $\mathbf{mintype}_\Delta([\,]) := [\,]$

$\mathbf{mintype}_\Delta([l_i = \varsigma(x_i : A)a_i]_{i \in 1 \dots n+1}) := A$ $\qquad$ $\mathbf{mintype}_\Delta(a.l \Leftarrow \varsigma(x : A)b) := A$

$\mathbf{mintype}_\Delta(a.l_j) := B_j$ when $\mathbf{mintype}_\Delta(a)$ is $[l_i : B_i]_{i \in 1 \dots n}$

3.3 Operational Semantics

The values of the simply-typed ς-calculus are plain objects. Selection and update are reduced by the following operational semantics rules where A stands for $[l_i : A_i]_{i \in 1 \dots n}$ and a stands for $[l_i = \varsigma(x_i : A)a_i]_{i \in 1 \dots n}$:

[4] Bold face is here used to distinguish the meta-level.

$$\begin{array}{lcl} a.l_j & \rightarrowtail & a_j\{a/x_j\} \\ a.l_j \Leftarrow \varsigma(x:A')u & \rightarrowtail & [l_j = \varsigma(x:A)u, l_i = \varsigma(x_i:A)a_i]_{i\in 1\ldots n, i\neq j} \end{array}$$

where $a_j\{a/x\}$ denotes the substitution of the variable x by the term a in term a_j.

The type A' used in the binder for updating the object a does not need to be equal to A but may be any supertype of it.

Subject reduction has been proved by Abadi and Cardelli [1]. However, reduction does not preserve minimum typing since $\mathbf{mintype}_\Delta\,(a.l_j \Leftarrow \varsigma(x:A')u)$ is (by definition) A' but this term reduces to a value of type A.

3.4 Example

The expressivity of the ς-calculus can be illustrated by the following example from Abadi and Cardelli [2] assuming that we have a type Num for numbers and that the simply-typed λ-calculus has been encoded:

$$\begin{array}{lll} \mathit{RomCell} & := & [\ \mathit{get} : \mathit{Num}\ ] \\ \mathit{PromCell} & := & [\ \mathit{get} : \mathit{Num}, \mathit{set} : \mathit{Num} \to \mathit{RomCell}\ ] \\ \mathit{PrivateCell} & := & [\ \mathit{get} : \mathit{Num}, \mathit{contents} : \mathit{Num}, \mathit{set} : \mathit{Num} \to \mathit{RomCell}\ ] \\ \mathit{myCell} : \mathit{PromCell} & := & [\ \mathit{get} = \varsigma(x : \mathit{PrivateCell})x.\mathit{contents}, \\ & & \quad \mathit{contents} = \varsigma(x : \mathit{PrivateCell})0, \\ & & \quad \mathit{set} = \varsigma(x : \mathit{PrivateCell})\lambda(n : \mathit{Num})x.\mathit{contents} \Leftarrow n\ ] \end{array}$$

RomCell is the type of read-only memory cells; the only action that we can perform on a *RomCell* is to read it (*get* method).

A *PromCell* is a memory cell which can be written once (*set* method), we can either read it now or write it and get a *RomCell*.

PrivateCell is a type used for implementation; it extends *PromCell* with a *contents* field which should not be seen from the outside.

The object *myCell* implemented as an object of type *PrivateCell* can be given the type *PromCell* thanks to subsumption.

4 Encoding of the simply-typed ς-calculus in the λΠ-calculus modulo

This section describes an encoding of the simply-typed ς-calculus given by a λΠm-context and a translation of ς-types, terms, and contexts. We want it to be shallow in the sense discussed in the introduction. However, the encoding described in the current section will only preserve typing and binding, since preserving reduction of a non terminating system cannot, of course, be achieved using a strongly-normalizing rewrite system. The associated rewrite system will be confluent and strongly normalizing, making type-checking of encoded terms decidable. In the next section, we will add a few rewrite rules in order to preserve reduction at the price of losing normalization.

This encoding is implemented as a translation tool [10] producing Dedukti terms from ς-terms and types.

4.1 Encoding of types

We assume given an infinite λΠ-type `label` with a decidable equality.

Unit, product, Σ-types, and Leibnitz equality can all be encoded in λΠm (they are special cases of inductive types, which are translated to λΠm by Coqine [4]) so we consider that they are available with the usual notations (respectively `unit`, $A \times B$, $\Sigma x : A.B$, and $=_A$). To avoid confusion with Leibnitz equality, we write $\equiv$ for the equality at meta-level.

4.1.1 Domains

Domains are lists of labels:

> $\texttt{domain} : \texttt{Type}.$
> $\texttt{nil} : \texttt{domain}.$
> $\texttt{cons} : \texttt{label} \to \texttt{domain} \to \texttt{domain}.$

We use the notation $[l_1; \dots; l_n]$ for $(\texttt{cons}\ l_1\ (\dots(\texttt{cons}\ l_n\ \texttt{nil})\dots))$.

We avoid assuming that our domains are duplicate-free and we instead consider proofs of membership of labels. The computational content of such a membership proof is relevant: it is a position in the list where the label appears. We simply call membership proofs *positions*:

> $\bullet \in \bullet : \texttt{label} \to \texttt{domain} \to \texttt{Type}.$
> $\texttt{at-head} : \Pi l.\Pi d.l \in \texttt{cons}\ l\ d.$
> $\texttt{in-tail} : \Pi l_1.\Pi l_2.\Pi d.l_1 \in d \to l_1 \in \texttt{cons}\ l_2\ d.$

Most functions in the encoding are defined by induction on positions.

We use the notation $d_1 \subset d_2$ as an abbreviation for $\Pi l.l \in d_1 \to l \in d_2$.

4.1.2 Object types

Types are encoded as sorted association lists. Sorting is done at translation time so we don't need an ordering on labels in the target language.

Formally, we declare the following type and terms:

> $\texttt{type} : \texttt{Type}.$
> $\texttt{typenil} : \texttt{type}.$
> $\texttt{typecons} : \texttt{label} \to \texttt{type} \to \texttt{type} \to \texttt{type}.$

The $\lambda\Pi$-term `type` should not be confused with the $\lambda\Pi$-term `Type`; the former is the $\lambda\Pi$ equivalent of ς-types and the latter is sort of all the $\lambda\Pi$-types.

A translation function $[\![\bullet]\!]$ from ς-types to $\lambda\Pi$-terms of type `type` is given by

$$[\![[l_i : A_i]_{i\in 1\dots n, l_1<\dots<l_n}]\!] := \texttt{typecons}\ l_1\ [\![A_1]\!]\ (\dots(\texttt{typecons}\ l_n\ [\![A_n]\!]\ \texttt{typenil})\dots)$$

For example, the types *RomCell*, *PromCell*, and *PrivateCell* defined in Section 3.4 are translated as follows:

$$\begin{array}{lcl}
[\![\mathit{RomCell}]\!] & \equiv & \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ \texttt{typenil} \\
[\![\mathit{PromCell}]\!] & \equiv & \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ (\\
 & & \quad \texttt{typecons}\ \mathit{set}\ [\![\mathit{Num} \to \mathit{RomCell}]\!] \\
 & & \quad\quad \texttt{typenil}) \\
[\![\mathit{PrivateCell}]\!] & \equiv & \texttt{typecons}\ \mathit{contents}\ [\![\mathit{Num}]\!]\ (\\
 & & \quad \texttt{typecons}\ \mathit{get}\ [\![\mathit{Num}]\!]\ (\\
 & & \quad\quad \texttt{typecons}\ \mathit{set}\ [\![\mathit{Num} \to \mathit{RomCell}]\!] \\
 & & \quad\quad\quad \texttt{typenil}))
\end{array}$$

4.1.3 Design choices

This encoding of ς-types as association lists is a bit under-specified: the type `type` does not impose unicity of label nor sorting. We know two ways to impose these two restrictions:

- We can add an extra argument to the `typecons` constructor, witnessing that the added label minors the elements in the tail of the list:

$$
\begin{array}{l}
\texttt{type} : \texttt{Type}.\\
\texttt{minors} : \texttt{label} \to \texttt{type} \to \texttt{Type}.\\
\texttt{typenil} : \texttt{type}.\\
\texttt{typecons} : \Pi l : \texttt{label}.\Pi A : \texttt{type}.\Pi B : \texttt{type}.\texttt{minors}\ l\ B \to \texttt{type}.\\
\texttt{minors-nil} : \Pi l : \texttt{label}.\texttt{minors}\ l\ \texttt{typenil}.\\
\texttt{minors-cons} : \Pi l.\Pi l'.\Pi A.\Pi B.\texttt{minors}\ l'\ B \to\\
\qquad\qquad\qquad l < l' \to \texttt{minors}\ l\ (\texttt{typecons}\ l'\ A\ B).
\end{array}
$$

  But this increases a lot the size of the translated types.
- It is also possible to quotient the association lists by a rule exchanging the order of entries and a rule removing duplicates:

$$
\begin{array}{lcl}
\texttt{typecons}\ l_1\ A_1\ (\texttt{typecons}\ l_2\ A_2\ B) & \hookrightarrow & \texttt{typecons}\ l_2\ A_2\ (\texttt{typecons}\ l_1\ A_1\ B).\\
\texttt{typecons}\ l\ A_1\ (\texttt{typecons}\ l\ A_2\ B) & \hookrightarrow & \texttt{typecons}\ l\ A_1\ B.
\end{array}
$$

  In order to preserve normalization, we have to guard the first rule by a condition like $l_2 < l_1$. Unfortunately, the resulting rewrite system becomes hard to keep confluent with definitions of functions on `type`. Moreover this requires an ordering on labels and the use of conditional rewriting which is not yet implemented in Dedukti.

The benefit from excluding unsorted association lists does not seem worth the drawbacks of these solutions hence we prefer to live with the existence of λΠ-terms of type `type` not coming from the encoding.

4.1.4 Domain and association

Since types are translated as association lists, we define the usual functions `assoc` and `dom` for respectively looking up an association and listing the domain:

$$
\begin{array}{lcl}
\texttt{dom} : \texttt{type} \to \texttt{domain}. & & \\
\texttt{dom}\ \texttt{typenil} & \hookrightarrow & \texttt{nil}.\\
\texttt{dom}\ (\texttt{typecons}\ l\ A\ B) & \hookrightarrow & \texttt{cons}\ l\ (\texttt{dom}\ B).\\
 & & \\
\texttt{assoc} : \Pi A : \texttt{type}.\Pi l : \texttt{label}.l \in \texttt{dom}\ A \to \texttt{type}. & & \\
\texttt{assoc}\ (\texttt{typecons}\ l\ A\ B)\ l\ (\texttt{at-head}\ l\ (\texttt{dom}\ B)) & \hookrightarrow & A.\\
\texttt{assoc}\ (\texttt{typecons}\ l_2\ A\ B)\ l_1\ (\texttt{in-tail}\ l_1\ l_2\ (\texttt{dom}\ B)\ p) & \hookrightarrow & \texttt{assoc}\ B\ l_1\ p.
\end{array}
$$

We abbreviate $\texttt{assoc}\ A\ l\ p$ as $A._p l$ or $A.l$ making the position p implicit.

4.1.5 Subtyping relations

The subtyping relation is defined by:

$$
\begin{array}{lcl}
\bullet \leq \bullet : \texttt{type} \to \texttt{type} \to \texttt{Type}. & & \\
A \leq \texttt{typenil} & \hookrightarrow & \texttt{unit}.\\
A \leq \texttt{typecons}\ l\ B\ C & \hookrightarrow & \Sigma p : l \in \texttt{dom}\ A.(A._p l =_{\texttt{type}} B) \times (A \leq C).
\end{array}
$$

where $=_{\texttt{type}}$ is the Leibnitz equality defined on `type`.

4.1.6 Properties of the subtyping relation

This subsection lists a few useful properties of the $\leq$ relation. These properties are provable directly in $\lambda\Pi$m, as opposed to the correctness of the translation of subtyping which will be addressed in Section 4.3.2. These proofs can be found at [10].

▶ **Lemma 1** (subtype-weakening). *The $\leq$ relation enjoys weakening; it means that in $\lambda\Pi$m, we can define a total function* `subtype-weakening` *of type* $\Pi A.\Pi B.\Pi l.\Pi C.A \leq B \to (\texttt{typecons}\ l\ A\ C) \leq B$.

Proof. Straightforward by induction on B (as explained previously, the function ***subtype-weakening*** is defined by two rewrite rules, one for $B \equiv$ `typenil` and another for $B \equiv$ `typecons ...`). ◀

▶ **Lemma 2** (subtype-refl). *The $\leq$ relation is reflexive; in $\lambda\Pi$m, we can define a total function* `subtype-refl` *of type* $\Pi A.A \leq A$.

Proof. By induction on A using the previous lemma. ◀

▶ **Lemma 3** (subtype-dom). *The dom function is compatible with $\leq$; in $\lambda\Pi$m, we can define a total function* `subtype-dom` *of type* $\Pi A.\Pi B.A \leq B \to \texttt{dom}\ B \subset \texttt{dom}\ A$.

Proof. By induction on B.

- base case is trivial (there is no rewrite rule for this case because it is an empty case)
- if B is `typecons` $l'\ B_1\ B_2$, we have some position $p' : l' \in \texttt{dom}\ A$ and $A \leq B_2$. For any l and any position $p : l \in \texttt{cons}\ l\ (\texttt{dom}\ B_2)$, either p is at head in which case $l \equiv l'$ and p' proves the goal, or p is in tail and we conclude using the induction hypothesis. ◀

▶ **Lemma 4** (subtype-assoc). *The assoc function is compatible with $\leq$; in $\lambda\Pi$m, we can define a total function* `subtype-assoc` *of type* $\Pi A.\Pi B.\Pi st : A \leq B.\Pi l.\Pi p : l \in \texttt{dom}\ B.B._p l =_{\texttt{type}} A._{\texttt{subtype-dom}\ A\ B\ st\ l\ p}\ l$.

Proof. By induction on B.

- base case is trivial
- if B is `typecons` $l'\ B_1\ B_2$, we have some position $p' : l' \in \texttt{dom}\ A$ such that $A._{p'} l' =_{\texttt{type}} B_1$ and $A \leq B_2$. For any l and any position $p : l' \in \texttt{cons}\ l\ (\texttt{dom}\ B_2)$,
  - either p is at head in which case $l \equiv l'$ and $B._p l \equiv B_1$. $A._{\texttt{subtype-dom}\ A\ B\ st\ l\ p} l' \equiv A._{p'} l' \equiv B_1$
  - or p is in tail in which case we conclude again using the induction hypothesis. ◀

▶ **Lemma 5** (subtype-trans). *The subtyping relation is transitive; in $\lambda\Pi$m, we can define a total function* `subtype-trans` *of type* $\Pi A.\Pi B.\Pi C.A \leq B \to B \leq C \to A \leq C$.

Proof. By induction on C, using `subtype-dom` and `subtype-assoc`. ◀

4.2 Encoding of terms

As we did for types, we define translation functions from terms and contexts of the simply-typed ς-calculus to terms and contexts of $\lambda\Pi$m.

These functions preserve typing in the sense that we can define, in $\lambda\Pi$m, a function `Expr` such that whenever the judgment $\Delta \vdash_\varsigma a : A$ is valid in the simply-typed ς-calculus, the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \texttt{Expr}\ [\![A]\!]$ is valid in $\lambda\Pi$m.

We define a $\lambda\Pi$m-context reflecting the syntax and the semantics of the ς-calculus. We start with concrete objects, we then define coercions reflecting the use of the subsumption rule. From these declarations, we define the $\lambda\Pi$m version of selection and update, and finally we give the translation function for terms.

4.2.1 Objects

`Expr` A represents the λΠm-type of well-typed objects of type A and `Meth` A B represents the λΠm-type of methods of A returning an object of type B.

We can declare `Expr` and `Meth`:

$$\begin{array}{l}\texttt{Expr} : \texttt{type} \to \texttt{Type}.\\ \texttt{Meth} : \texttt{type} \to \texttt{type} \to \texttt{Type}.\end{array}$$

Unfortunately, we cannot define `Expr` directly by some nil and cons constructors, as we did for types, because a sublist of a well-typed object is not well-typed.

We call a sublist of a well-typed object of type A, defined on some set of labels d, a *preobject* of type (A, d).

Formally, we define a λΠm-type `Preobj` A d by the following declarations:

$$\begin{array}{l}\texttt{Preobj} : \texttt{type} \to \texttt{domain} \to \texttt{Type}.\\ \texttt{prenil} : \Pi A.\texttt{Preobj}\ A\ \texttt{nil}.\\ \texttt{precons} : \Pi A.\Pi d.\Pi l.\Pi p : l \in \texttt{dom}\ A.\\ \qquad\qquad \texttt{Meth}\ A\ A._p l \to \texttt{Preobj}\ A\ d \to \texttt{Preobj}\ A\ (\texttt{cons}\ l\ d).\end{array}$$

With preobjects at hand, we can define objects of type A:

$$\texttt{Obj}\ A \hookrightarrow \texttt{Preobj}\ A\ (\texttt{dom}\ A).$$

and expressions of type B are objects of a type A, subtype of B:

$$\texttt{Expr}\ B \hookrightarrow \Sigma A : \texttt{type}.(\texttt{Obj}\ A) \times (A \leq B).$$

Since the subtyping relation is reflexive, we can inject objects into expressions:

$$\begin{array}{l}\texttt{expr-of-obj} : \Pi A.\texttt{Obj}\ A \to \texttt{Expr}\ A.\\ \texttt{expr-of-obj}\ a \hookrightarrow (A, a, \texttt{subtype-refl}\ A).\end{array}$$

We would like to define `Meth` A B as `Expr` $A \to$ `Expr` B to end this set of definitions but then the negative occurrence of `Expr` would be a source of non-termination.

We solve this problem by adding axioms stating that `Meth` A B is equivalent to `Expr` $A \to$ `Expr` B:

$$\begin{array}{ll}\texttt{Eval-meth} & : \Pi A.\Pi B.\texttt{Meth}\ A\ B \to \texttt{Expr}\ A \to \texttt{Expr}\ B.\\ \texttt{Make-meth} & : \Pi A.\Pi B.(\texttt{Expr}\ A \to \texttt{Expr}\ B) \to \texttt{Meth}\ A\ B.\end{array}$$

The key point here is that `Eval-meth` and `Make-meth` will freeze reduction. For example the translation of a looping ς-term like $[l = \varsigma(x : [l : []])x.l].l$ will be a term whose normalization will freeze at an occurrence of the pattern `Eval-meth` A B (`Make-meth` A B f) a which will not be matched by any rewrite rule.

To get a reduction-preserving encoding, we just have to add some rewrite rules; either the rule `Eval-meth` A B (`Make-meth` A B f) $a \hookrightarrow f\ a$ or the following one `Meth` A $B \hookrightarrow$ `Expr` $A \to$ `Expr` B (and `Eval-meth` and `Make-meth` both reduce to identity).

4.2.2 Coercions

Implicit subtyping cannot be expressed in λΠm because each λΠ-term has at most one type modulo β and rewriting. Hence we cannot simply rewrite any type A to any of its subtypes or supertypes; rewriting is oriented but conversion is symmetric.

Since we cannot use implicit subtyping, we have to define some explicit coercion operation to be used instead of the subsumption typing rule.

These coercions are actually very easy to define thanks to our definition of `Expr` and Lemma 5; if a is an object of type A subtype of B seen as an expression of type B, seeing a as an expression of type C supertype of B only requires a proof of $A \leq C$ which may be obtained by transitivity of $\leq$:

$$\begin{array}{l}\texttt{coerce} : \Pi B : \texttt{type}.\Pi C : \texttt{type}.B \leq C \to \texttt{Expr}\ B \to \texttt{Expr}\ C.\\ \texttt{coerce}\ B\ C\ st_{BC}\ (A, a, st_{AB}) \hookrightarrow (A, a, \texttt{subtype-trans}\ st_{AB}\ st_{BC}).\end{array}$$

We use the notation $a \uparrow_A^B$ for the term `coerce` $A\ B\ st\ a$ of type `Expr` B, leaving the subtyping proof implicit.

4.2.3 Operational semantics

The `select` and `update` functions explore the object until they find the corresponding method and either return it or rebuild another object.

Their definitions follow the definitions of `Expr` and `Obj`; they work recursively on the `Preobj` structure using auxiliary functions called `preselect` and `preupdate`. These functions operate on a preobject of type (A, d) and are defined by induction on a position $p : l \in d$ which can be converted to a position of type $l \in \texttt{dom}\ A$ thanks to the following lemma:

▶ **Lemma 6** (preobj-subset). *Preobjects are defined on subsets of the domain: in* $\lambda\Pi m$*, we can define a total function* `preobj-subset` *of type* $\Pi A.\Pi d.\texttt{Preobj}\ A\ d \to d \subset \texttt{dom}\ A$.

Proof. Straightforward by induction on d. ◀

The definition of update is straightforward:

$$\begin{array}{l}\texttt{preupdate} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\Pi po : \texttt{Preobj}\ A\ d.\\ \qquad\qquad \texttt{Meth}\ A\ A._{\texttt{preobj-subset}\ A\ d\ po\ l\ p} l \to \texttt{Preobj}\ A\ d.\\ \texttt{obj-update} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Obj}\ A.\\ \texttt{update} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Meth}\ A\ A._p l \to \texttt{Expr}\ A.\\ \\ \texttt{preupdate}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m'\ po)\ m\\ \quad \hookrightarrow \texttt{precons}\ A\ d\ l\ p'\ m\ po.\\ \texttt{preupdate}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po)\ m\\ \quad \hookrightarrow \texttt{precons}\ A\ d\ l'\ p'\ m'\ (\texttt{preupdate}\ A\ d\ l\ p\ po\ m).\\ \\ \texttt{obj-update}\ A\ l\ p\ a\ m \quad \hookrightarrow \quad \texttt{preupdate}\ A\ (\texttt{dom}\ A)\ l\ p\ a\ m.\end{array}$$

The `obj-update` function can be used to update a method of an object of type A; if we want to update an expression of type B where $A \leq B$, we only have at hand a method of type `Meth` $B\ A.l$ (for some l) where `obj-update` needs a `Meth` $A\ A.l$. This can be solved by a substitution of the $self$ variable by its coercion $self \uparrow_A^B$ in the method body, which is easy to write as $(\texttt{Make-meth}\ A\ A.l\ ((\lambda(self : \texttt{Expr}\ A)\ (\texttt{Eval-meth}\ B\ A.l\ m\ (self \uparrow_A^B)))))$. Hence we can define `update` as follows:

$$\begin{array}{l}\texttt{update}\ B\ l\ p\ (A, a, st)\ m\\ \quad \hookrightarrow (A,\\ \qquad \texttt{obj-update}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ p\ )\ a\\ \qquad\quad (\texttt{Make-meth}\ A\ A.l\ (\lambda(self : \texttt{Expr}\ A)\ (\texttt{Eval-meth}\ B\ A.l\ m\ (self \uparrow_A^B)))),\\ \qquad st).\end{array}$$

Selection is a bit more subtle because we need both the selected method, which is found by inductively destructing the object, and the full object which should be substituted for the self variable. The `preselect` function doesn't return an object but the method associated with the label. The `select` function duplicates its argument a, one copy is passed to `preselect` and the other is used with the returned method to build a blocked redex using the `Eval-meth` axiom.

$$
\begin{array}{l}
\texttt{preselect} : \Pi A.\Pi d.\Pi l.\Pi p : l \in d.\texttt{Preobj}\ A\ d \to \texttt{Meth}\ A\ (A._p l).\\
\texttt{obj-select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Obj}\ A \to \texttt{Meth}\ A\ (A._p l).\\
\texttt{select} : \Pi A.\Pi l.\Pi p : l \in \texttt{dom}\ A.\texttt{Expr}\ A \to \texttt{Expr}\ A._p l.\\
\\
\texttt{preselect}\ A\ (\texttt{cons}\ l\ d)\ l\ (\texttt{at-head}\ l\ d)\ (\texttt{precons}\ A\ d\ l\ p'\ m\ po)\\
\quad \hookrightarrow m.\\
\texttt{preselect}\ A\ (\texttt{cons}\ l'\ d)\ l\ (\texttt{in-tail}\ l\ l'\ d\ p)\ (\texttt{precons}\ A\ d\ l'\ p'\ m'\ po)\\
\quad \hookrightarrow \texttt{preselect}\ A\ d\ l\ p\ po.\\
\\
\texttt{obj-select}\ A\ l\ p\ a \ \hookrightarrow\ \texttt{preselect}\ A\ (\texttt{dom}\ A)\ l\ p\ a.\\
\\
\texttt{select}\ B\ l\ p\ (A, a, st) \ \hookrightarrow\ \texttt{Eval-meth}\ A\ A._p\ l\\
\qquad\qquad\qquad\qquad (\texttt{obj-select}\ A\ l\ p\ a)\ (A, a, st).
\end{array}
$$

4.2.4 Translation function for expressions

We now have all we need to define a translation function from simply-typed ς-terms to λΠm.

The same ς-term a may have to be translated to different λΠ-terms of different types because λΠm lacks subtyping and subsumption. Hence we have to parameterize our translation function by the targeted type A of a in the ς-calculus. Fortunately, it is enough to define the translation function for the minimum type of a, written $[\![a]\!]_\Delta$. We can then define the general translation function for type A as $[\![a]\!]_{\Delta,A} := [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ where the proof of $[\![\mathbf{mintype}_\Delta\,(a)]\!] \leq [\![A]\!]$ is computed by a meta-level[5] function **decide-subtype** (omitted here).

The $[\![\bullet]\!]_\Delta$ function, the $[\![\bullet]\!]_{\Delta,A}$ function and the translation function for methods are mutually defined by:

$$
\begin{array}{l}
[\![a]\!]_{\Delta,A} := [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}\\
\\
[\![[l_i = \varsigma(x : A)a_i]_{i\in 1\ldots n, l_1<\ldots<l_n}]\!]_\Delta\\
\quad := \texttt{expr-of-obj}\ (\\
\qquad \texttt{precons}\ [\![A]\!]\ [l_2; \ldots; l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A._{p_1} l_1}\ (\\
\qquad \ldots (\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A._{p_n} l_n}\ \texttt{prenil}\ [\![A]\!])))\\
\\
[\![a.l]\!]_\Delta := \texttt{select}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]\ l\ p\ [\![a]\!]_\Delta\\
[\![a.l \Leftarrow \varsigma(x : A)b]\!]_\Delta := \texttt{update}\ [\![A]\!]\ l\ p\ [\![a]\!]_{\Delta,A}\ [\![\varsigma(x : A)b]\!]_{\Delta,A._p l}\\
\\
[\![\varsigma(x : A)b]\!]_{\Delta,B}\\
\quad := \texttt{Make-meth}\ [\![A]\!]\ [\![B]\!]\ (\lambda x : \texttt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta,x:A),B})
\end{array}
$$

[5] The function **decide-subtype** is easy to define at the meta-level but could also be defined in λΠm.

The positions p_i and p in this encoding can be computed for any well-typed ς-term : p_i is the ith position (p_1 is $\texttt{at-head}\ l_1\ [l_2;\dots;l_n]$, p_2 is $\texttt{in-tail}\ l_2\ l_1\ (\texttt{at-head}\ l_2\ [l_3;\dots;l_n])$, p_n is $\texttt{in-tail}\ l_n\ l_1\ (\dots(\texttt{in-tail}\ l_n\ l_{n-1}\ (\texttt{at-head}\ l_n\ [\,])\,)\dots)$, and p is the p_i such that l is l_i).

The translation of the binding operation of our source language (the ς binder) is done by a binding operation in the target language (the λ binder). This technique is generally known as Higher-Order Abstract Syntax (HOAS) [29].

We can now compute the translation of our example term *myCell*. We translate a term a by an object of type $[\![\mathbf{mintype}_\Delta\ (a)]\!]$ seen as an expression of the required type. In this case, $\mathbf{mintype}_\Delta\ (myCell)$ is *PrivateCell* and the required type is *PromCell*.

$$
\begin{aligned}
&[\![myCell]\!]_{\Delta,PromCell}\\
&\quad\equiv [\![myCell]\!]_\Delta \uparrow^{[\![PromCell]\!]}_{[\![\mathbf{mintype}_\Delta(myCell)]\!]}\\
&\quad\equiv [\![myCell]\!]_\Delta \uparrow^{[\![PromCell]\!]}_{[\![PrivateCell]\!]}\\
&\quad\equiv ([\![PrivateCell]\!],\\
&\qquad \texttt{precons}\ [\![PrivateCell]\!]\ [get; set]\ contents\ p_1\\
&\qquad\quad [\![\varsigma(x : PrivateCell)0]\!]_{\Delta,Num}(\\
&\qquad\quad \texttt{precons}\ [\![PrivateCell]\!]\ [set]\ get\ p_2\\
&\qquad\qquad [\![\varsigma(x : PrivateCell)x.contents]\!]_{\Delta,Num}(\\
&\qquad\qquad \texttt{precons}\ [\![PrivateCell]\!]\ [\,]\ set\ p_3\\
&\qquad\qquad\quad [\![\varsigma(x : PrivateCell)\lambda(n : Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell}(\\
&\qquad\qquad\quad \texttt{prenil}\ [\![PrivateCell]\!])))),\\
&\qquad \textbf{decide-subtype}\ [\![PrivateCell]\!]\ [\![PromCell]\!])
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x : PrivateCell)0]\!]_{\Delta,Num}\\
&\quad\equiv \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num]\!]\\
&\qquad (\lambda x : \texttt{Expr}\ [\![PrivateCell]\!].[\![0]\!]_{(\Delta,x:PrivateCell),Num})
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x : PrivateCell)x.contents]\!]_{\Delta,Num}\\
&\quad\equiv \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num]\!]\\
&\qquad (\lambda x : \texttt{Expr}\ [\![PrivateCell]\!].\texttt{select}\ [\![Num]\!]\ contents\ p_1\ x)
\end{aligned}
$$

$$
\begin{aligned}
&[\![\varsigma(x : PrivateCell)\lambda(n : Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell}\\
&\quad\equiv \texttt{Make-meth}\ [\![PrivateCell]\!]\ [\![Num \to RomCell]\!]\\
&\qquad (\lambda x : \texttt{Expr}\ [\![PrivateCell]\!].[\![\lambda(n : Num)x.contents \Leftarrow n]\!]_{\Delta,Num\to RomCell})
\end{aligned}
$$

As expected, the translation of the looping ς-term $[l = \varsigma(x : [l : [\,]])x.l].l$ normalizes to an instance of the pattern $\texttt{Eval-meth}\ A\ B\ (\texttt{Make-meth}\ A\ B\ f)\ a$:

$$
\begin{aligned}
&[\![[l : [\,]]]\!]_\emptyset && \equiv \texttt{typecons}\ l\ \texttt{typenil}\ \texttt{typenil}\\
&[\![x.l]\!]_{x:[l:[\,]]} && \equiv \texttt{select}\ [\![[l : [\,]]]\!]_{x:[l:[\,]]}\ l\ p_1\ x\\
&[\![\varsigma(x : [l : [\,]])x.l]\!]_{\emptyset,[]} && \equiv \texttt{Make-meth}\ [\![[l : [\,]]]\!]_\emptyset\ [\,]\ (\lambda x : \texttt{Expr}\ [\![[l : [\,]]]\!]_\emptyset.[\![x.l]\!]_{x:[l:[\,]]})\\
&[\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_\emptyset && \equiv ([\![[l : [\,]]]\!]_\emptyset,\\
&\qquad\qquad && \quad \texttt{precons}\ [\![[l : [\,]]]\!]_\emptyset\ [\,]\ l\ p_1\ [\![\varsigma(x : [l : [\,]])x.l]\!]_{\emptyset,[]}\\
&\qquad\qquad && \qquad (\texttt{prenil}\ [\![[l : [\,]]]\!]_\emptyset),\\
&\qquad\qquad && \quad \texttt{subtype-refl}\ [\![[l : [\,]]]\!]_\emptyset)
\end{aligned}
$$

$$
\begin{aligned}
[\![[l = \varsigma(x : [l : [\,]])x.l].l]\!]_\emptyset \quad &\equiv \texttt{select}\ [\![[l : [\,]]]\!]_\Delta\ l\ p_1\ [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_\emptyset \\
&\hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_\emptyset\ [\,]\ l \\
&\quad (\texttt{obj-select}\ [\![[l : [\,]]]\!]_\emptyset\ l\ p_1 \\
&\quad\quad (\texttt{precons}\ \ldots)) \\
&\quad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_\emptyset \\
&\hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_\emptyset\ [\,]\ l \\
&\quad (\texttt{preselect}\ [\![[l : [\,]]]\!]_\emptyset\ [l]\ l\ p_1 \\
&\quad\quad (\texttt{precons}\ \ldots)) \\
&\quad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_\emptyset \\
&\hookrightarrow \texttt{Eval-meth}\ [\![[l : [\,]]]\!]_\emptyset\ [\,]\ l \\
&\quad\quad (\texttt{Make-meth}\ [\![[l : [\,]]]\!]_\emptyset\ [\,] \\
&\quad\quad\quad (\lambda x : \texttt{Expr}\ [\![[l : [\,]]]\!]_\emptyset.[\![x.l]\!]_{x:[l:[\,]]})) \\
&\quad\quad\quad\quad [\![[l = \varsigma(x : [l : [\,]])x.l]]\!]_\emptyset
\end{aligned}
$$

4.3 Properties of the encoding

Let Γ_0 be the $\lambda\Pi$m-context composed of the declarations and rewrite rules presented previously in this section. We investigate properties of the rewrite system $\mathbf{R}_0$ associated with Γ_0 and of translated ς-terms in contexts of the form Γ_0, Λ where Λ is a $\lambda\Pi$-context (a $\lambda\Pi$m-context without rewrite rule) so the rewrite system associated with Γ_0, Λ is $\mathbf{R}_0$.

The proofs in this section are done at the meta-level and are pen-and-paper proofs.

4.3.1 Normalization and confluence

The rewrite system $\mathbf{R}_0$ is strongly normalizing because recursive calls are performed on strict subterms and variables of left-hand sides are never applied in the right-hand side. It is also confluent because it is left-linear and normalizing [27].

In order to be extra-confident in these properties, we implemented the definitions of Γ_0 in the Calculus of Inductive Constructions, which is known to be strongly normalizing and confluent [15], and type-checked this implementation with Coq.

Our code is available at `http://sigmaid.gforge.inria.fr`. However this translation to Coq uses axioms (`Meth`, `Make-meth`, and `Eval-meth`) which are *a priori* not provable in Coq.

4.3.2 Preservation of the subtyping relation by the translation

In this subsection we prove that our translation of types preserves subtyping: given two ς-types A and B, we have $A <: B$ if and only if $[\![A]\!] \leq [\![B]\!]$. The proof requires some intermediate results we detail below.

▶ **Lemma 7.** *If* $l \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$ *then* $l \equiv l_j$ *for some* $j \in 1 \ldots n$.

Proof. Trivial by induction on the position of type $l \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$. ◀

▶ **Lemma 8.** *If* $j \in 1 \ldots n$, *then* $l_j \in \texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$.

Proof. Without loss of generality, we assume that $l_1 > \ldots > l_n$. $\texttt{dom}\ [\![[l_i : A_i]_{i\in 1\ldots n}]\!]$ is $[l_n; \ldots; l_1]$. We prove that $l_j \in [l_n; \ldots; l_1]$ by induction on n:

- case $n \equiv 0$: the hypothesis $j \in 1 \ldots n$ is a contradiction.

- case $n \equiv p+1$: if $j \equiv p+1$ then `at-head` j $[l_p; \dots; l_1]$ proves $l_j \in [l_{p+1}; \dots l_1]$ else $j \in 1 \dots p$ so by induction hypothesis, $l_j \in [l_p; \dots l_1]$ thus $l_j \in [l_{p+1}; \dots; l_1]$ by `in-tail`. ◀

▶ **Lemma 9.** *If $j \in 1 \dots n$, then $[\![[l_i : A_i]_{i \in 1 \dots n}]\!].{}_{pos} l_j \equiv [\![A_j]\!]$ where pos is the proof of the previous lemma.*

Proof. This is trivial by following the same steps as in the previous lemma. ◀

▶ **Theorem 10.** *For every type A and B, if $\Delta \vdash_\varsigma A <: B$ then $[\![A]\!] \leq [\![B]\!]$.*

Proof. We proceed by induction on the derivation of $\Delta \vdash_\varsigma A <: B$, there are three cases:

- case (subtype)
  A is some $[l_i : A_i]_{i \in 1 \dots n+m}$ with $B \equiv [l_i : A_i]_{i \in 1 \dots n}$. Without loss of generality, we may assume $l_n < l_{n-1} < \dots < l_2 < l_1$. We proceed by induction on n:
  - case $n \equiv 0$: $[\![B]\!] \equiv$ `typenil` hence $[\![A]\!] \leq [\![B]\!]$.
  - case $n \equiv p+1$: $[\![B]\!] \equiv$ `typecons` l_{p+1} $[\![A_{p+1}]\!]$ $[\![[l_i : A_i]_{i \in 1 \dots p}]\!]$.

    $$\begin{aligned}
    &[\![A]\!] \leq [\![B]\!] \\
    &\quad \equiv [\![A]\!] \leq \texttt{typecons}\ l_{p+1}\ [\![A_{p+1}]\!]\ [\![[l_i : A_i]_{i \in 1 \dots p}]\!] \\
    &\quad \equiv \Sigma pos : l_{p+1} \in \texttt{dom}\ [\![A]\!]. \\
    &\qquad \left([\![A]\!].{}_{pos} l_{p+1} =_{\texttt{type}} [\![A_{p+1}]\!]\right) \times \left([\![A]\!] \leq [\![[l_i : A_i]_{i \in 1 \dots p}]\!]\right)
    \end{aligned}$$

    pos and the equality proof are given by Lemma 8 and Lemma 9. The proof of $[\![A]\!] \leq [\![[l_i : A_i]_{i \in 1 \dots p}]\!]$ is given by the induction hypothesis.
- case (refl)
  This is trivial by Lemma 2.
- case (trans)
  This is trivial by Lemma 5. ◀

▶ **Theorem 11.** *The translation function on types is injective: if $[\![A]\!] =_{type} [\![B]\!]$ then $A \equiv B$.*

Proof. A type and its encoding have the same size hence A and B have the same size. The proof is by induction on this common size; both cases are trivial. ◀

▶ **Theorem 12.** *For every type A and B, well-formed in context Δ, if $[\![A]\!] \leq [\![B]\!]$ then $\Delta \vdash_\varsigma A <: B$.*

Proof. By induction on the size n of $B := [l_i : B_i]_{l_1 > \dots > l_n}$.

- case $n \equiv 0$: $B \equiv [\,]$ hence $\Delta \vdash_\varsigma A <: B$.
- case $n \equiv p+1$: $[\![B]\!] \equiv$ `typecons` l_{p+1} $[\![B_{p+1}]\!]$ $[\![[l_i : B_i]_{l_1 > \dots > l_p}]\!]$. Our hypothesis simplifies to:

  $$\begin{aligned}
  &[\![A]\!] \leq [\![B]\!] \\
  &\quad \equiv [\![A]\!] \leq \texttt{typecons}\ l_{p+1}\ [\![B_{p+1}]\!]\ [\![[l_i : B_i]_{l_1 > \dots > l_p}]\!] \\
  &\quad \equiv \Sigma pos : l_{p+1} \in \texttt{dom}\ [\![A]\!]. \\
  &\qquad \left([\![A]\!].{}_{pos} l_{p+1} =_{\texttt{type}} [\![B_{p+1}]\!]\right) \times \left([\![A]\!] \leq [\![[l_i : B_i]_{l_1 > \dots > l_p}]\!]\right)
  \end{aligned}$$

By induction hypothesis, A is of the form $[l_i : B_i; l'_j : A_j]_{i \in 1 \dots p, j \in 1 \dots m+1}$. From the lemmata and the injectivity theorem, we get $l_{p+1} \equiv l'_j$ and $A_j \equiv B_{p+1}$ for some $j \in 1 \dots m+1$. By renaming the l's, we can choose $j \equiv m+1$ and we get $A \equiv [l_i : B_i; l'_j : A_j]_{i \in 1 \dots p, j \in 1 \dots m}$ so $\Delta \vdash_\varsigma A <: B$ by rule (subtype). ◀

4.3.3 Type preservation

We want to prove the following type preservation theorem:

▶ **Theorem 13.** *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, then the encoded judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \mathit{Expr}\ [\![A]\!]$, is valid in λΠm.*

For this, we first need to define $[\![\Delta]\!]$:

$$\begin{array}{ll} [\![\emptyset]\!] & := \Gamma_0 \\ [\![\Delta, x : A]\!] & := [\![\Delta]\!], x : \mathtt{Expr}\ [\![A]\!] \end{array}$$

Since the translation function $[\![\bullet]\!]_{\Delta,A}$ is recursively defined together with $[\![\bullet]\!]_\Delta$ and the translation function for methods, we need lemmata to relate these three functions:

▶ **Lemma 14.** *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, and, in λΠm, the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \mathit{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$ is valid, then so is the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_{\Delta,A} : \mathit{Expr}\ [\![A]\!]$.*

Proof. From $\Delta \vdash_\varsigma a : A$ we get, by minimality, $\Delta \vdash_\varsigma \mathbf{mintype}_\Delta\,(a) <: A$ hence $[\![\mathbf{mintype}_\Delta\,(a)]\!] \leq [\![A]\!]$ by Theorem 10. Therefore $[\![a]\!]_{\Delta,A} \equiv [\![a]\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]}$ has type $\mathtt{Expr}\ [\![A]\!]$. ◀

▶ **Lemma 15.** *If, in λΠm, the judgment $[\![\Delta]\!], x : \mathit{Expr}\ [\![A]\!] \vdash_d [\![b]\!]_{(\Delta,x:A),B} : \mathit{Expr}\ [\![B]\!]$ is valid, then so is the judgment $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta,B} : \mathit{Meth}\ [\![A]\!]\ [\![B]\!]$.*

Proof. x doesn't occur free in $[\![B]\!]$ because it is a closed term. Hence we can type the λ-abstraction with an arrow type: $[\![\Delta]\!] \vdash_d (\lambda x : \mathtt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta,x:A),B}) : \mathtt{Expr}\ [\![A]\!] \to \mathtt{Expr}\ [\![B]\!]$.

Therefore $[\![\varsigma(x : A)b]\!]_{\Delta,B} \equiv \mathtt{Make\text{-}meth}\ [\![A]\!]\ [\![B]\!]\ (\lambda x : \mathtt{Expr}\ [\![A]\!].[\![b]\!]_{(\Delta,x:A),B})$ has type $\mathtt{Meth}\ [\![A]\!]\ [\![B]\!]$. ◀

▶ **Theorem 16.** *If, in the simply typed ς-calculus, the judgment $\Delta \vdash_\varsigma a : A$ is valid, then the judgment $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \mathit{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$ is valid in λΠm.*

Proof. By minimality, $\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta\,(a)$. We proceed by induction on this typing derivation; we have one case for each typing rule in the simply-typed ς-calculus:

- case (var): a is a variable x appearing in Δ and $\mathbf{mintype}_\Delta\,(a) \equiv \mathbf{mintype}_\Delta\,(x) \equiv \Delta(x)$. By definition of $[\![\Delta]\!]$, $x \in [\![\Delta]\!]$ and $[\![\Delta]\!](x) \equiv \mathtt{Expr}\ [\![\Delta(x)]\!] \equiv \mathtt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
- case (obj): a is $[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}$ with $\mathbf{mintype}_\Delta\,(a) \equiv A \equiv [l_i : A_i]_{l_1<\ldots<l_n}$.

  $$\begin{array}{rl} [\![a]\!]_\Delta & \equiv [\![[l_i = \varsigma(x_i : A)\ a_i]_{l_1<\ldots<l_n}]\!]_\Delta \\ & \equiv \mathtt{expr\text{-}of\text{-}obj}\ (\\ & \qquad \mathtt{precons}\ [\![A]\!]\ [l_2; \ldots; l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A.p_1 l_1}\ (\\ & \qquad \ldots (\mathtt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \mathtt{prenil}\ [\![A]\!]))) \end{array}$$

  The term $\mathtt{expr\text{-}of\text{-}obj}$ has type $\mathtt{Obj}\ [\![A]\!] \to \mathtt{Expr}\ [\![A]\!]$ so we just need to check that $\mathtt{precons}\ [\![A]\!]\ [l_2; \ldots; l_n]\ l_1\ p_1\ [\![\varsigma(x : A)a_1]\!]_{\Delta,A.p_1 l_1}\ (\ldots(\mathtt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta,A.p_n l_n}\ \mathtt{prenil}\ [\![A]\!]))$ has type $\mathtt{Obj}\ [\![A]\!]$.

- To compute $\mathtt{Obj}\ [\![A]\!]$, we first compute $\mathtt{dom}\ [\![A]\!]$:

  $$\begin{array}{rl} \mathtt{dom}\ [\![A]\!] & \equiv \mathtt{dom}\ [\![[l_i : A_i]_{l_1<\ldots<l_n}]\!] \\ & \equiv \mathtt{dom}\ (\mathtt{typecons}\ l_1\ [\![A_1]\!]\ (\ldots(\mathtt{typecons}\ l_n\ [\![A_n]\!]\ \mathtt{typenil})\ldots)) \\ & \equiv [l_1; \ldots; l_n] \end{array}$$

  hence $\mathtt{Obj}\ [\![A]\!] \equiv \mathtt{Preobj}\ [\![A]\!]\ (\mathtt{dom}\ [\![A]\!]) \equiv \mathtt{Preobj}\ [\![A]\!]\ [l_1; \ldots; l_n]$.

- We show by induction that each built preobject is well-typed with the expected type. For all $i \in 1 \dots n$,
  $[\![\Delta]\!] \vdash_d \quad \texttt{precons}\ [\![A]\!]\ [l_{i+1}; \dots; l_n]\ l_i\ p_i\ [\![\varsigma(x : A)a_i]\!]_{\Delta, A.{}_{p_i} l_i}\ ($
  $\dots (\texttt{precons}\ [\![A]\!]\ [\,]\ l_n\ p_n\ [\![\varsigma(x : A)a_n]\!]_{\Delta, A.{}_{p_n} l_n}\ \texttt{prenil}\ [\![A]\!]))$
  $: \texttt{Preobj}\ [\![A]\!]\ [l_i; \dots; l_n]$
  This is trivial by decreasing recursion on i.

Finally $[\![\Delta]\!] \vdash_d [\![[l_i = \varsigma(x_i : A)\ a_i]_{l_1 < \dots < l_n}]\!]_\Delta : \texttt{Expr}\ [\![A]\!]$.

- case (select): a is of the form $a'.l_j$ with $j \in 1 \dots n$ and $\Delta \vdash_\varsigma a' : A'$ where $A' := [l_i : A_i]_{i \in 1 \dots n}$. Without loss of generality, we can assume that A' is the minimal type of a'[6]: $\mathbf{mintype}_\Delta\,(a') \equiv [l_i : A_i]_{i \in 1 \dots n}$ so $\mathbf{mintype}_\Delta\,(a) \equiv A_j$.
  Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]$ hence by Lemma 9, $[\![\mathbf{mintype}_\Delta\,(a')]\!].{}_p l_j \equiv [\![A_j]\!] \equiv [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
  Moreover, $\Delta \vdash_\varsigma [\![a']\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]$ by induction hypothesis thus $[\![a]\!]_\Delta \equiv \texttt{select}\ [\![\mathbf{mintype}_\Delta\,(a')]\!]\ l_j\ p\ [\![a']\!]_\Delta$ has type $\texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
- case (update): a is of the form $a'.l_j \Leftarrow \varsigma(x : A)b$ with $j \in 1 \dots n$, $\Delta \vdash_\varsigma a' : A$, and $\Delta, x : A \vdash_\varsigma b : A_j$ where $A \equiv [l_i : A_i]_{i \in 1 \dots n}$.
  By induction hypothesis and Lemma 14, $[\![\Delta]\!] \vdash_d [\![a']\!]_{\Delta, A} : \texttt{Expr}\ A$. By Lemma 15, $[\![\Delta]\!] \vdash_d [\![\varsigma(x : A)b]\!]_{\Delta, A_j} : \texttt{Meth}\ [\![A]\!]\ [\![A_j]\!]$.
  Like in the previous case, Lemma 8 gives us a position $p : l_j \in \texttt{dom}\ [\![A]\!]$ and by Lemma 9, $[\![A]\!].{}_p l_j \equiv [\![A_j]\!]$.
  Hence $[\![a]\!]_\Delta \equiv \texttt{update}\ [\![A]\!]\ l_j\ p\ [\![a']\!]_{\Delta, A}\ [\![\varsigma(x : A)b]\!]_{\Delta, A_j}$ has type $[\![A]\!] \equiv [\![\mathbf{mintype}_\Delta\,(a)]\!]$.
- case (subsume): The only possible instantiation of the subsumption rule which derives a minimum typing is the trivial case

$$\frac{\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta\,(a) \qquad \Delta \vdash_\varsigma \mathbf{mintype}_\Delta\,(a) <: \mathbf{mintype}_\Delta\,(a)}{\Delta \vdash_\varsigma a : \mathbf{mintype}_\Delta\,(a)}\ (\text{subsume})$$

  In this case, our goal is exactly the induction hypothesis
  $[\![\Delta]\!] \vdash_d [\![a]\!]_\Delta : \texttt{Expr}\ [\![\mathbf{mintype}_\Delta\,(a)]\!]$.

◀

From this and Lemma 14, we have proved Theorem 13.

4.3.4 Semantics preservation and consistency

Semantics preservation is not ensured because our rewrite system is strongly normalizing and the simply-typed ς-calculus is not.

However, we may want the following weaker result:

▶ **Statement 1.** *If* $\Delta \vdash_\varsigma a : A$ *and* $a \rightarrowtail a'$ *then* $[\![a]\!]_{\Delta, A} =_{\texttt{Expr}\ [\![A]\!]} [\![a']\!]_{\Delta, A}$ *is inhabited in context* $[\![\Delta]\!]$.

In the case where a is a selection $a \equiv a''.l$, $[\![a]\!]_{\Delta, A}$ reduces to an instance of the pattern $\texttt{Eval-meth}\ B\ C\ (\texttt{Make-meth}\ B\ C\ f)\ b$ such that $[\![a']\!]_{\Delta, A} \equiv f\ b$.

Hence we would need

$$\texttt{reduce-meth} : \Pi B.\Pi C.\Pi f.\Pi b.\texttt{Eval-meth}\ B\ C\ (\texttt{Make-meth}\ B\ C\ f)\ b =_{\texttt{Expr}\ B} f\ b$$

[6] This comes from the proof of minimality in [1] (Propositions 4.1.1-1 to 4.1.1-4); a minimal typing judgment can be derived by allowing subsumption only before the (update) and (obj) rules.

as an additional axiom. Unfortunately, it would be inconsistent with our encoding so Statement 1 is hopeless. The following inconsistency result has been proved in Coq [10]:

▶ **Theorem 17.** *For any label l, the type*

$$\left(\Pi B.\Pi C.\Pi f.\Pi b.\mathit{Eval\text{-}meth}\ B\ C\ (\mathit{Make\text{-}meth}\ B\ C\ f)\ b =_{\mathit{Expr}\ C} f\ b\right) \to ([\,] =_{\mathit{type}} [l : [\,]])$$

is inhabited.

Proof. From an expression, we can extract the type of the underlying object:

$$\begin{array}{l}\texttt{underlying-type} : \Pi B.\texttt{Expr}\ B \to \texttt{type}.\\ \texttt{underlying-type}\ B\ (A, a, st) \hookrightarrow A.\end{array}$$

- Let A_0 be the type $[l : [\,]]$ and a_0 an inhabitant of $\texttt{Expr}\ A_0$ (for instance $a_0 : \texttt{Expr}\ A_0 := [l = [\,]]$). $t_0 := a_0 \uparrow^{[\,]}_{A_0}$ is an inhabitant of $\texttt{Expr}\ [\,]$ which we can distinguish from the empty expression $[\,]$ because they have different underlying types.
- We define a function $\texttt{swap} : \texttt{Expr}\ [\,] \to \texttt{Expr}\ [\,]$ returning an expression different from its argument:

$$\begin{array}{l}\texttt{swap-aux} : \texttt{type} \to \texttt{Expr}\ [\,].\\ \texttt{swap-aux}\ \texttt{typenil} \hookrightarrow t_0.\\ \texttt{swap-aux}\ (\texttt{typecons}\ l'\ B\ C) \hookrightarrow [\,].\\ \texttt{swap} : \texttt{Expr}\ [\,] \to \texttt{Expr}\ [\,].\\ \texttt{swap}\ b \hookrightarrow \texttt{swap-aux}\ (\texttt{underlying-type}\ b).\end{array}$$

- We remark that $\texttt{Expr}\ A_0$ is isomorphic to $\texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$:
  - We can define a function $\texttt{elim-A}_0 : \texttt{Expr}\ A_0 \to \texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$ by

    $$\texttt{elim-A}_0\ [l = \varsigma(x)f(x)] := f$$

  - and a function $\texttt{intro-A}_0 : (\texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]) \to \texttt{Expr}\ A_0$ by

    $$\texttt{intro-A}_0\ f := [l = \varsigma(x)f(x)]$$

- let $E_0 : \texttt{Expr}\ A_0 \to \texttt{Expr}\ [\,]$ be the function defined by $E_0\ a := \texttt{swap}(\texttt{elim-A}_0\ a)$. Then $b_0 : \texttt{Expr}\ [\,] := E_0\ (\texttt{intro-A}_0\ E_0)$ is such that we can prove, using the `reduce-meth` axiom, $b_0 =_{\texttt{Expr}\ [\,]} \texttt{swap}\ b_0$, hence
  $\texttt{underlying-type}\ (\texttt{swap}\ b_0) = \texttt{underlying-type}\ (\texttt{swap}\ (\texttt{swap}\ b_0))$ but $\texttt{swap}\ b_0$ is either $[\,]$ or t_0 and we get $([\,] =_{\texttt{type}} [l : [\,]])$ in both cases. This last step is actually an adaption of the proof of Cantor's theorem. ◀

Consistency is hard to define in λΠm because we have not even defined anything looking like the false proposition. Consistency is to be defined relatively to a given logic. However, we probably never want $([\,] =_{\texttt{type}} [l : [\,]])$ to be inhabited.

5 Shallow, non-terminating encoding

In this section, we trade strong-normalization for a shallow encoding.

5.1 Modified rewrite system

In order to get a shallow encoding, we have to add the following rewrite rules:

$$\begin{aligned}
&\texttt{Meth}\ A\ B \hookrightarrow \texttt{Expr}\ A \to \texttt{Expr}\ B.\\
&\texttt{Eval-meth}\ A\ B\ m\ \hookrightarrow m.\\
&\texttt{Make-meth}\ A\ B\ f \hookrightarrow f.
\end{aligned}$$

From this, the `reduce-meth` axiom can trivially be proved so we need to change our encoding a bit to forbid the proof of Theorem 17. We do this by disabling the extraction of underlying type and the distinction between objects and expressions. Instead of defining $\texttt{Expr}\ B$ as $\Sigma A : \texttt{type}.(\texttt{Obj}\ A) \times (A \leq B)$, we rewrite $\texttt{Expr}\ A$ to $\texttt{Obj}\ A$ and change the definitions of the functions that destructed expressions: `update`, `select`, and `coerce`:

$$\begin{aligned}
&\texttt{Expr}\ A \hookrightarrow \texttt{Obj}\ A.\\
&\texttt{expr-of-obj}\ a \hookrightarrow a.\\
&\texttt{update}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet)\ \bullet\\
&\quad\hookrightarrow \texttt{obj-update}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet)\ \bullet.\\
&\texttt{update}\ B\ l\ p\ (\texttt{coerce}\ A\ B\ st\ a)\ m\\
&\quad\hookrightarrow \texttt{coerce}\ A\ B\ st\ \big(\texttt{update}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ l\ p)\ a\\
&\qquad(\lambda(self).m\ (self \uparrow_A^B))\big).\\
&\texttt{select}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet)\\
&\quad\hookrightarrow \texttt{obj-select}\ \bullet\ \bullet\ \bullet\ (\texttt{precons}\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet\ \bullet).\\
&\texttt{select}\ B\ l\ p\ (\texttt{coerce}\ A\ B\ st\ a)\\
&\quad\hookrightarrow \texttt{select}\ A\ l\ (\texttt{subtype-dom}\ A\ B\ st\ l\ p)\ a.\\
&\texttt{coerce}\ B\ C\ st_{BC}\ (\texttt{coerce}\ A\ B\ st_{AB}\ a)\\
&\quad\hookrightarrow \texttt{coerce}\ A\ C\ (\texttt{subtype-trans}\ st_{AB}\ st_{BC})\ a
\end{aligned}$$

The `coerce` function is not total anymore because it does not reduce on values but only when applied to another coercion. It is a constructor of `Expr` with some computational behaviour; we call such constructors *smart* constructors. The bullets in the previous rules defining `update` and `select` represent the most general pattern that make these rules well-typed. The idea here is simply that `update` and `select` are defined by pattern matching on the object, which is either a value or a coercion. We don't need rules for the `prenil` case because there is no label to select or update in that case.

We call Γ_1 this new $\lambda\Pi$m-context and $\mathbf{R}_1$ the new rewrite system. We believe that $\mathbf{R}_1$ is confluent because the non-orthogonal part reflects the simply-typed ς-calculus known to be confluent [2], but have not formally checked it. However, $\mathbf{R}_1$ is not expected to be (strongly or even weakly) normalizing. Hence Dedukti will type-check encoded object programs only if they are well-typed but may not answer on non-terminating terms[7].

5.2 Semantics preservation

Proofs of the theorems of Section 4 are unchanged because they did not rely on the definitions of `update`, `select`, and `coerce`. The new encoding has the additional property of semantics preservation:

[7] Actually it will terminate because
- conversion check, which triggers reduction, only occurs in types;
- non-termination only occurs at the object level;
- there is no dependent type involving objects coming from our encoding.

▶ **Theorem 18.** *If* $\Delta \vdash_\varsigma a : A$ *and* $a \rightarrowtail a'$ *then* $[\![a]\!]_{\Delta,A} \hookrightarrow^+ [\![a']\!]_{\Delta,A}$.

To prove this theorem, we first need two lemmata: stability of the encoding by substitution and unicity of subtyping proofs.

▶ **Lemma 19.** *The translation function is stable by substitution:*
$[\![a]\!]_{(\Delta_1,x:B),A}\{[\![b]\!]_{(\Delta_1,\Delta_2),B}/x\} \equiv [\![a\{b/x\}]\!]_{(\Delta_1,\Delta_2),A}$.

Proof. This comes from the fact that binding operation is preserved by the encoding. This can be proved by induction on a. ◀

▶ **Lemma 20.** *Unicity of subtype proofs: if* st_1 *and* st_2 *both have type* $[\![A]\!] \leq [\![B]\!]$ *then* $st_1 =_{[\![A]\!]\leq[\![B]\!]} st_2$.

This lemma justifies our use of implicit subtype proofs in the notation $\bullet \uparrow^{[\![\bullet]\!]}_{[\![\bullet]\!]}$.

Proof. Unicity of subtype proofs comes from the fact that $[\![A]\!]$ is duplicate-free. We don't use, however, the fact that $[\![B]\!]$ is duplicate-free and prove this theorem for any β[8] of type `type`: if st_1 and st_2 both have type $[\![A]\!] \leq \beta$ then $st_1 =_{[\![A]\!]\leq\beta} st_2$.

We proceed by induction on β.

- base case: $\beta \equiv$ `typenil`
  $[\![A]\!] \leq \beta \equiv [\![A]\!] \leq$ `typenil` $\equiv$ `unit`. The type `unit` has only one inhabitant so $st_1 =_{[\![A]\!]\leq\beta} st_2$.
- inductive case: $\beta \equiv$ `typecons` $l\ \beta_1\ \beta_2$
  A is some $[l_i : A_i]_{l_1<\ldots<l_n}$.
  By definition of $\leq$,

$$[\![A]\!] \leq \texttt{typecons}\ l\ \beta_1\ \beta_2 \equiv \Sigma p : l \in [l_1; \ldots; l_n].([\![A]\!].p l =_{\texttt{type}} \beta_1) \times ([\![A]\!] \leq \beta_2)$$

  But there is only one $p : l \in [l_1; \ldots; l_n]$ because the l_is are different. Let us call it p_0.
  $[\![A]\!] \leq$ `typecons` $l\ \beta_1\ \beta_2$ is isomorphic to $([\![A]\!].p_0 l =_{\texttt{type}} \beta_1) \times ([\![A]\!] \leq \beta_2)$.
  The left type $[\![A]\!].p_0 l =_{\texttt{type}} \beta_1$ has at most one inhabitant thanks to Hedberg Theorem [21] because equality on `type` is decidable; the right type $[\![A]\!] \leq \beta_2$ has only one element by induction hypothesis so $st_1 =_{[\![A]\!]\leq\beta} st_2$. ◀

We can now prove Theorem 18:

Proof. The simply-typed ς-calculus enjoys subject-reduction [2] so $\Delta \vdash_\varsigma a' : A$. From the type-preservation theorem, $[\![a]\!]_{\Delta,A}$ and $[\![a']\!]_{\Delta,A}$ have type `Expr` $[\![A]\!]$ in context $[\![\Delta]\!]$.

We proceed by induction on the operational semantics definition; there are two cases:

- case (select): $a \rightarrowtail a'$ is an instance of $a''.l_j \rightarrowtail a_j\{a''/x_j\}$
  with $a'' := [l_i = \varsigma(x_i : A'')a_i]_{i\in1\ldots n}$ and $A'' := [l_i : A_i]_{i=1\ldots n}$.
  So $a \equiv a''.l_j$ and $a' \equiv a_j\{a''/x_j\}$.
  We look at the minimum types of a and a':
  - $\mathbf{mintype}_\Delta\,(a'') \equiv A'' \equiv [l_i : A_i]_{i\in1\ldots n}$ so $\mathbf{mintype}_\Delta\,(a) \equiv \mathbf{mintype}_\Delta\,(a''.l_j) \equiv A_j$
  - We call A' the minimum type of a', by minimality we know that $\Delta \vdash_\varsigma A' <: A_j$.

[8] We use the greek letter β here to distinguish the ς-term B and the λΠ-term β which abstracts $[\![B]\!]$.

$[\![a'']\!]_\Delta$ is of the form $(\dots(\texttt{precons}\ [\![A'']\!]\ [l_{j+1};\dots;l_n]\ l_j\ [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j}\dots)\ )$, we abbreviate it as α.

$$\begin{array}{lll}
[\![a]\!]_\Delta & \equiv & \texttt{select}\ [\![A'']\!]\ l_j\ p\ \alpha \\
& \hookrightarrow & \texttt{obj-select}\ [\![A'']\!]\ l_j\ p\ \alpha\ \alpha \\
& \hookrightarrow & \texttt{preselect}\ [\![A'']\!]\ [l_1;\dots;l_n]\ l_j\ p\ \alpha\ \alpha \\
& \hookrightarrow^* & [\![\varsigma(x_j : A'')a_j]\!]_{\Delta,A_j}\ \alpha \\
& \equiv & \left(\lambda x_j : \texttt{Expr}\ [\![A'']\!].[\![a_j]\!]_{(\Delta,x_j:A''),A_j}\right)\ \alpha \\
& \longrightarrow_\beta & [\![a_j]\!]_{(\Delta,x_j:A''),A_j}\{[\![a'']\!]_\Delta/x\}
\end{array}$$

Hence, by Lemma 19, we get exactly $[\![a]\!]_\Delta \hookrightarrow^+ [\![a']\!]_{\Delta,A_j}$.
Finally,

$$\begin{array}{ll}
[\![a]\!]_{\Delta,A} & \equiv [\![a]\!] \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a)]\!]} \\
& \hookrightarrow^+ [\![a']\!]_{\Delta,A_j} \uparrow^{[\![A]\!]}_{[\![A_j]\!]} \\
& \equiv \left([\![a']\!]_\Delta \uparrow^{[\![A_j]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]}\right) \uparrow^{[\![A]\!]}_{[\![A_j]\!]} \\
& \hookrightarrow [\![a']\!]_\Delta \uparrow^{[\![A]\!]}_{[\![\mathbf{mintype}_\Delta(a')]\!]} \\
& \equiv [\![a']\!]_{\Delta,A}
\end{array}$$

- case (update): this case is very similar to the previous one, only simpler because we don't need to use the substitution lemma. ◀

6 Conclusion

We defined an embedding of the simply-typed ς-calculus to λΠm and implemented it in Dedukti as a compiler named sigmaid (SIGMA-calculus In Dedukti) [10]. This implementation has been tested on the following original examples from Abadi and Cardelli:

- encoding of the simply-typed λ-calculus,
- encoding of booleans,
- memory cells.

Despite non-termination of the ς-calculus, we managed to translate it in a very shallow fashion by means of two encodings: a normalizing one and a semantics-preserving one.

This embedding is a starting point for other shallow embeddings of typed object oriented calculi with subtyping:

- Beside common extensions for object type systems (polymorphism, variance annotations, type operators), we are especially interested in extending this work to object type systems with dependent types in order to study dependently-typed objects combining computational methods and logical methods which depend upon them and prove their specifications. These logical methods would be proofs taking benefits of the mechanisms of object oriented programming.
- We would also like to encode class-based calculi like Featherweight Java [25] in λΠm in order to compare the encoded versions of structural subtyping and class-based subtyping.

Acknowledgment. We would like to thank our colleague Ali Assaf for the fruitful discussions which led to this implementation of subtyping in λΠm. We also thank Alan Schmitt and anonymous referees for their comments that helped us to improve this article.

References

1 Martín Abadi and Luca Cardelli. A theory of primitive objects: Untyped and first-order systems. In *TACS'94, Theoretical Aspects of Computing Sofware*, pages 296–320, 1994.

2 Martín Abadi and Luca Cardelli. *A Theory of Objects*. Monographs in Computer Science. Springer New York, 1996.

3 Ali Assaf and Guillaume Burel. Holide. `https://www.rocq.inria.fr/deducteam/Holide/index.html`.

4 Mathieu Boespflug and Guillaume Burel. CoqInE : Translating the calculus of inductive constructions into the λΠ-calculus modulo. In *International Workshop on Proof Exchange for Theorem Proving*, 2012.

5 Mathieu Boespflug, Quentin Carbonneaux, and Olivier Hermant. The lambda-pi-calculus modulo as a universal proof language. In *International Workshop on Proof Exchange for Theorem Proving*, 2012.

6 Mathieu Boespflug, Quentin Carbonneaux, Olivier Hermant, and Ronan Saillard. Dedukti: a universal proof checker. `http://dedukti.gforge.inria.fr`.

7 Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1/2):108–133, November 1999.

8 Guillaume Burel. A shallow embedding of resolution and superposition proofs into the λπ-calculus modulo. In *International Workshop on Proof Exchange for Theorem Proving*, 2013.

9 Raphaël Cauderlier. Focalide. `https://www.rocq.inria.fr/deducteam/Focalide`.

10 Raphaël Cauderlier. Sigmaid. `http://sigmaid.gforge.inria.fr`.

11 Alberto Ciaffaglione, Luigi Liquori, and Marino Miculan. Reasoning about object-based calculi in (co)inductive type theory and the theory of contexts. *J. Autom. Reason.*, 39(1):1–47, July 2007.

12 Horatiu Cirstea, Claude Kirchner, and Luigi Liquori. Matching Power. In *Proceedings of RTA'2001*, Lecture Notes in Computer Science. Springer-Verlag, May 2001.

13 Horatiu Cirstea, Luigi Liquori, and Benjamin Wack. Rewriting calculus with fixpoints: Untyped and first-order systems. In *TYPES*, volume 3085. Springer, 2003.

14 Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. Manual distributed as documentation of the Maude system, Computer Science Laboratory, SRI International. `http://maude.csl.sri.com/manual`, January 1999.

15 The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.

16 Denis Cousineau and Gilles Dowek. Embedding pure type systems in the lambda-pi-calculus modulo. In Simona Ronchi Della Rocca, editor, *TLCA*, volume 4583 of *LNCS*, pages 102–117. Springer, 2007.

17 David Delahaye, Damien Doligez, Frédéric Gilbert, Pierre Halmagrand, and Olivier Hermant. Zenon Modulo: When Achilles Outruns the Tortoise using Deduction Modulo. In *Logic for Programming Artificial Intelligence and Reasoning (LPAR)*, volume 8312 of *LNCS/ARCoSS*. Springer, 2013.

18 Kathleen Fisher, Furio Honsell, and John C. Mitchell. A lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1:3–37, 1994.

19 J. Nathan Foster and Dimitrios Vytiniotis. A theory of featherweight java in isabelle/hol. *Archive of Formal Proofs*, March 2006. `http://afp.sf.net/entries/FeatherweightJava.shtml`, Formal proof development.

20 Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *J. ACM*, 40(1):143–184, January 1993.

21 Michael Hedberg. A coherence theorem for martin-löf's type theory. *J. Functional Programming*, pages 4–8, 1998.

22 Ludovic Henrio and Florian Kammüller. A mechanized model of the theory of objects. In MarcelloM. Bonsangue and EinarBroch Johnsen, editors, *Formal Methods for Open Object-Based Distributed Systems*, volume 4468 of *Lecture Notes in Computer Science*, pages 190–205. Springer Berlin Heidelberg, 2007.

23 Ludovic Henrio, Florian Kammüller, Bianca Lutz, and Henry Sudhof. Locally nameless sigma calculus. *Archive of Formal Proofs*, April 2010. `http://afp.sf.net/entries/Locally-Nameless-Sigma.shtml`, Formal proof development.

24 focalize-devel@inria.fr. Focalize. `http://focalize.inria.fr`.

25 Atsushi Igarashi, Benjamin Pierce, and Philip Wadler. Featherweight java – a minimal core calculus for java and gj. In *ACM Transactions on Programming Languages and Systems*, pages 132–146, 1999.

26 Julian Mackay, Hannes Mehnert, Alex Potanin, Lindsay Groves, and Nicholas Cameron. Encoding featherweight java with assignment and immutability using the coq proof assistant. In *Workshop on Formal Techniques for Java-like Programs*, FTfJP'12, pages 11–19, New York, NY, USA, 2012. ACM.

27 Fritz Müller. Confluence of the lambda calculus with left-linear algebraic rewriting. *Information Processing Letters*, 41(6):293–299, 1992.

28 R. P. Nederpelt, J. H. Geuvers, and R. C. de Vrijer. *Selected papers on Automath*. Elsevier, Amsterdam, 1994.

29 Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Programming Language Design and Implementation*, 1988.

30 Frank Pfenning and Carsten Schurmann. System description: Twelf – a meta-logical framework for deductive systems. In *International Conference on Automated Deduction (CADE-16)*, pages 202–206. Springer-Verlag LNAI, 1999.

31 Benjamin C. Pierce and David N. Turner. Simple type-theoretic foundations for object-oriented programming. *Journal of Functional Programming*, 4(2):207–247, April 1994.

32 Jan Zwanenburg. *Object-Oriented Concepts and Proof Rules: Formalization in Type Theory and Implementation in Yarrow*. PhD thesis, Eindhoven University of Technology, 1999.

Typeful Normalization by Evaluation

Olivier Danvy[1], Chantal Keller[2], and Matthias Puech[3]

1 **Department of Computer Science, Aarhus University, Denmark**
`danvy@cs.au.dk`
2 **Microsoft Research Cambridge, United Kingdom and**
Microsoft Research – Inria Joint Centre, France
`Chantal.Keller@inria.fr`
3 **School of Computer Science, McGill University, Montreal, Canada**
`puech@cs.mcgill.ca`

Abstract

We present the first typeful implementation of Normalization by Evaluation for the simply typed λ-calculus with sums and control operators:

- we guarantee type preservation and η-long (modulo commuting conversions), β-normal forms using only Generalized Algebraic Data Types in a general-purpose programming language, here OCaml; and
- we account for finite sums and control operators with Continuation-Passing Style.

Our presentation takes the form of a typed functional pearl. First, we implement the standard NbE algorithm for the implicational fragment in a typeful way that is correct by construction. We then derive its continuation-passing counterpart, in call-by-value and call-by-name, that maps a λ-term with sums and call/cc into a CPS term in normal form, which we express in a typed, dedicated syntax. Beyond showcasing the expressive power of GADTs, we emphasize that type inference gives a smooth way to re-derive the encodings of the syntax and typing of normal forms in Continuation-Passing Style.

1998 ACM Subject Classification D.1.1 Applicative (Functional) Programming, F.3.2 Semantics of Programming Languages

Keywords and phrases Normalization by Evaluation, Generalized Algebraic Data Types, Continuation-Passing Style, partial evaluation

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.72

1 Introduction

A normalization function need not be *reduction-based* and rely on reiterated one-step reduction, according to some strategy, until a normal form is obtained, if any. It can be *reduction-free*, and, as pioneered by Berger and Schwichtenberg [15], one can obtain it by composing an evaluation function (towards a non-standard domain of values) together with a left-inverse reification function (towards normal forms). The concept of this 'normalization by evaluation' (the term is due to Schwichtenberg [14]) arose in a variety of contexts: intuitionistic logic [1, 21, 48], proof theory [15], program extraction [12], category theory [22, 53], models of computation [40], program transformation [28], partial evaluation [24, 33], etc. [27]. It has been vigorously studied since [8, 11, 43, 56].

A recent example of the power of normalization by evaluation (NbE for short) lies in the new reduction engine developed by Boespflug *et al.* [16, 17] for the Coq proof assistant.[1] It

[1] The command `Compute` in Coq triggers a call to Coq's reduction engine.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 72–88
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

improves the efficiency of proofs by reflection by an order of magnitude [4], and in Gonthier's words [38], proofs by reflection are what made it possible to prove the four-color theorem.

In this article, we propose a formalization of NbE for the simply-typed λ-calculus with sums and control operators in the general-purpose language OCaml, in such a way that the type system guarantees two key properties:

- NbE produces *normal forms*: the resulting term is in η-long, β-normal form;
- NbE is *type-preserving*: the type of the resulting term is the same as the type of the source term.

These are guaranteed by OCaml's subject reduction, provided that we stay in its purely functional, terminating fragment (which is a meta-argument).

To address sums and control operators, we use Continuation-Passing Style (CPS for short) in a novel way: we show that CPS-transforming the standard, typed NbE algorithm not only leaves room for these constructs, but also lets us derive a syntax of CPS normal forms and its typing rules. The resulting NbE program maps typed λ-terms to typed CPS normal forms.

Throughout, we use *Generalized Algebraic Data Types* (GADTs for short), a generalization of ML algebraic data types that allows a fine control on the return type of their constructors [19, 54]. We use them not only to represent the types and the well-typed terms of the simply-typed λ-calculus, but more interestingly to relate them to the types of values and of normal forms. The use of GADTs inherently limits us to simply typed objects languages, but our main motivation is to give a clean presentation of NbE for non-trivial aspects of such languages.

Faithful formalizations of NbE in direct style already exist in languages with dependent types like Coq or Agda [6, 13, 36, 42]. These complex languages already rely on an implementation of normalization for type-checking, which is precisely what we embark on implementing. Instead, we chose a general-purpose programming language featuring only weak-head evaluation and type inference. Our programming language of discourse is OCaml, which now provides support for GADTs [37], but we could have adopted any other functional programming language with this feature, e.g., Haskell, as partly done by Danvy, Rhiger, and Rose with type classes [32]. (We write "partly" because the "long" aspect of the resulting η-long, β-normal forms needed a meta-argument.) Alternatively, we could have used any other functional language by encoding GADTs [55] or by using some indirect representation of terms as functions ("finally tagless", phantom types, etc.) [18, 47]. Using GADTs, we can keep representing syntax as algebraic data types, as customary. This conservative design enables a methodology where the code is left essentially unchanged and only the types are refined.

Outline. The remainder of this article is an incremental, literate programming exposition of our implementation in the form of a typed functional pearl.[2] We first recall and motivate our starting points: the representation of types, terms, and values in OCaml, the standard NbE algorithm for the implicational fragment in direct style, and GADTs (Section 2). We annotate the standard NbE program to obtain a typeful implementation in direct style, that we put to use for the partial evaluation of `printf` directives (Section 3). We CPS-transform this typeful implementation, obtaining another typeful implementation that yields typed normal forms in CPS (Section 4). This continuation-passing typeful implementation is ready to be extended with sums and control operators.

[2] We will however allow ourselves to pedagogically reorder some code snippets. The full code is currently available at `cs.mcgill.ca/~puech/typeful.ml`.

2 Background

2.1 Deep and shallow embeddings

Since NbE manipulates types, terms and values of the λ-calculus, we need to represent all of them in our programming language of discourse, OCaml. When embedding a language into another, one has essentially two options: a deep embedding or a shallow embedding.

- In a deep embedding, to each construct of the language corresponds a constructor of a data type; we have access to the structure of terms, and we can define functions over them by structural recursion. The types and terms of the λ-calculus can be encoded this way in OCaml: one data type representing simple types (featuring an uninstantiated base type)

```
type tp =
  | Base                                    (* Uninstantiated base type *)
  | Arr of tp * tp
```

  and another one for terms. For concision, we use a weak (or parametric) Higher-Order Abstract Syntax representation of binders [20] (HOAS for short), where variables belong to an abstract type, and are introduced by OCaml functions:[3]

```
type tm =
  | Var of x
  | Lam of (x → tm)
  | App of tm * tm
and x                 (* The variable namespace, uninstantiated for now *)
```

- In a shallow embedding, we directly use OCaml constructs to represent constructs in the object language: we lose structural recursion, but we enjoy the property that two $\beta\eta$-equivalent values in OCaml are observationally equal. The values of the λ-calculus can be encoded this way: functions are represented as a universal function space, and we reuse OCaml variables and applications syntax nodes.

```
type base                           (* Base type, uninstantiated for now *)
type vl =
  | VFun of (vl → vl)
  | VBase of base
```

▶ **Example 1.** The term $\lambda f x.\, f\ x$ is represented as `Lam (fun f → Lam (fun x → App (Var f, Var x)))` in the deep encoding of terms, and as `VFun (fun (VFun f) → VFun (fun x → f x))` in the shallow encoding of values.

2.2 Normalization by Evaluation

NbE normalizes deeply embedded terms by going through a shallow embedding: an evaluation function maps a deep term to its shallow counterpart, which is then reified back into a deep term. Since $\beta\eta$-equivalent terms are indistinguishable at the shallow level, reification has to

[3] First-order presentations like de Bruijn indices are also common, and have been showed to be isomorphic to weak HOAS [7]. This way, we avoid Kripke-like parametrization of the target language, and we separate concerns better.

pick the same representative for two $\beta\eta$-equivalent terms (in practice, the η-long β-normal form, which implies that the result is in normal form).

First, the evaluation function maps application nodes App in the deep encoding into shallow, OCaml applications:

```
let rec eval : tm → vl = function
  | Var x → x
  | Lam f → VFun (fun x → eval (f x))
  | App (m, n) → match eval m with
    | VFun f → f (eval n)
    | VBase b → failwith "Unidentified␣Functional␣Object"
```

In the second case, variables are substituted with their value; to this end, we must instantiate their namespace with the type of values, allowing the constructor Var to quote values into terms:[4]

```
and x = vl
```

The expressible values `vl` are shallow values, i.e., weak-head normal forms. The second step consists in reifying them back into an algebraic language of deep terms, or *normal forms* `nf`, that can be inspected by pattern matching:

```
and nf =
  | NLam of (y → nf)
  | NAt of at
and at =
  | AApp of at * nf
  | AVar of y
and y
```

To proscribe the representation of β-redexes, we follow the tradition and stratify the syntax into *normal forms* `nf` (λ-abstractions) and *atoms* `at` (applications). Type `y` is the uninstantiated domain of target variables.

We then define the reification function `reify`, taking a value and its type to a normal form, together with its symmetric counterpart, `reflect`. They can be seen as performing a two-level η-expansion at the given type [30]. This η-expansion stops when encountering a value of the uninstantiated base type, which means that values of base type actually stand for atoms:

```
and base = Atom of at
```

In other words, atoms are the intersection of the set of shallow values and deep terms, reflecting the fact that values contain both functions and atoms.

All of this leads us to the usual definition of reification and reflection:

```
let rec reify : tp → vl → nf = fun a v → match a, v with
  | Arr (a, b), VFun f → NLam (fun x → reify b (f (reflect a (AVar x))))
  | Base, VBase v → let (Atom r) = v in NAt r
  | a, v → failwith "type␣mismatch"
```

[4] One could object that this instantiation of the domain of variables takes us away from weak HOAS. However, it is only necessary for the source language of `eval`, and a commodity to avoid more verbose solutions like de Bruijn indices or explicit parametricity in type `x` [51].

```
and reflect : tp → at → vl = fun a r → match a with
  | Arr (a, b) → VFun (fun x → reflect b (AApp (r, reify a x)))
  | Base → VBase (Atom r)
```

Finally, NbE maps a term together with its type to a normal form, by composing evaluation and reification:

```
let nbe : tp → tm → nf = fun a m → reify a (eval m)
```

Notice that exceptions might be triggered at runtime if the given term and type do not match. In Section 3, we solve this problem by statically enforcing this match, thanks to GADTs.

2.3 GADTs in OCaml

The recent introduction of Generalized Algebraic Data Types [19, 54] in OCaml [37] makes it syntactically possible to constrain type parameters for the return type of the constructors of a data type, which enables, e.g., to write tagless interpreters. Let us illustrate GADTs with the problem of formatting strings *à la* printf in a type-safe way, following Kiselyov [46] and OCaml's recent Printf module; it will serve as a running example in this article.

What is the type of the `printf` function in the C programming language? *A priori* it is dependent: the number of arguments depends on the structure of the first argument, the *formatting directive*. The first author proposed a solution based on polymorphism [25], encoding the formatting directive algebraically as a sequence of literal strings and typed placeholders (written `"%d"`, `"%s"`, etc. in C) and encoding it with CPS. GADTs provide language support for this encoding. Let us introduce the type of formatting directives, respectively indexed by α, the final type returned by `printf`, and β, the expected type of `printf` when applied only to the directive

```
type (α, β) directive =
```

These two types coincide when the directive consists only of a literal: no extra argument is then required. We thus explicitly mention the annotation after the argument in the constructor type:

```
  | Lit : string → (α, α) directive
```

When the directive is a placeholder, we add an argument to the expected type of `printf` (these constructors take no arguments):

```
  | String : (α, string → α) directive
  | Int : (α, int → α) directive
```

Finally, the sequence of two directives threads the initial and final types, much like function composition (and indeed the first author's encoding for sequence was function composition in CPS):

```
  | Seq : (β, γ) directive * (α, β) directive → (α, γ) directive
```

After spreading some syntactic sugar, let us try out this definition with an example directive (`"%d␣*␣%s␣=␣%d␣in␣%s"` in C):

```
let (^^) a b = Seq (a, b) and (!) x = Lit x and d = Int and s = String
let ex_directive : (α, int → string → int → string → α) directive =
  d ^^ !"␣*␣" ^^ s ^^ !"␣=␣" ^^ d ^^ !"␣in␣" ^^ s
```

The type reflects the structure of the formatting directive: an integer is expected, and then a string, and then an integer, and then a string, and then the result is whatever it needs to be.

Now, all `printf` needs to do is to map a directive into a usual OCaml primitive function. We first define it in CPS, and then we apply it to the initial continuation `print_string`, which will emit the formatted string eventually:

```
let rec kprintf : type a b. (a, b) directive → (string → a) → b =
  function
  | Lit s → fun k → k s
  | Int → fun k x → k (string_of_int x)
  | String → fun k x → k (string_of_string x)
  | Seq (f,g) → fun k → kprintf f (fun v → kprintf g (fun w → k (v^w)))
let printf dir = kprintf dir print_string
```

Function `string_of_string` here is the identity. Compared to the previous solutions [5, 25], which used one polymorphic function per abstract-syntax constructor of the formatting directive, the dispatch among the constructors is grouped, thanks to the GADTs.

Our test directive yields a type-safe printing command:

```
(* prints "6 * 9 = 42 in base 13" *)
let () = printf ex_directive 6 "9" 42 "base␣13"
```

3 Typeful Normalization by Evaluation in Direct Style

Thanks to GADTs, we can decorate the algebraic data types of terms and normal forms with their types, such that only well-typed ones can be represented. This way, the NbE algorithm of Section 2.2 can ensure statically that: *i)* no exception is triggered at runtime; *ii)* well-typed terms are mapped to well-typed normal forms; and *iii)* η-long normal forms are produced (in addition to being β-normal, which is new [32]). We then illustrate this normalizer with a partial evaluator that is guaranteed to preserve the type of the programs it specializes.

3.1 Evaluation

It is a standard use of GADTs to index terms – deep or shallow – by the OCaml type of their interpretation. First, values can be indexed as follows (we will come back to the definition of type `base` later on):

```
type α vl =
  | VFun : (α vl → β vl) → (α → β) vl
  | VBase : base → base vl
```

Note that this type definition does not respect the positivity condition, in the sense of, e.g., Coq, because there is a negative occurrence of `vl`. It is, however, stratified in the sense of Abella [35], i.e., its type parameter gets syntactically smaller. Thus, it forms a valid inductive definition. Ditto for terms (the same remark as in Section 2.2 applies to type α `x`):

```
and α x = α vl
type α tm =
  | Lam : (α x → β tm) → (α → β) tm
  | App : (α → β) tm * α tm → β tm
  | Var : α x → α tm
```

The evaluation function now has type α `tm` $\to$ α `vl`, ensuring type preservation:

```
let rec eval : type a. a tm → a vl = function
  | Var x → x
  | Lam f → VFun (fun x → eval (f x))
  | App (m, n) → let VFun f = eval m in f (eval n)
```

Because the match between types and terms is ensured statically, there is no need for any exception as in Section 2.2. Otherwise, the code remains the same.

▶ Remark. Evaluation could also have been *tagless*, and thus more efficient [17], i.e., we could have defined directly `type` α `vl =` α. We did not do so to be consistent with Section 4. Also, the *finally tagless* approach [18] can alternatively implement typeful NbE without GADTs [47], but it requires significant changes compared to the previous, untyped version: there, evaluation and reification are not recursive functions but define the syntax of terms and types.

3.2 Reification

In the same way, we can index atoms and normal forms with the type of their interpretations:

```
and α nf =
  | NLam : (α y → β nf) → (α → β) nf
  | NAt : base at → base nf
and α at =
  | AApp : (α → β) at * α nf → β at
  | AVar : α y → α at
and α y
```

The variable domain α `y` is left uninstantiated. In addition to being β-normal, the restriction of the NAt coercion to a base type guarantees that terms of this data type are also η-long [3].

We then need to statically relate our deep types `tp` with these annotations. To this end, we can index them by the OCaml type of their denotation:

```
type α tp =
  | Base : base tp
  | Arr : α tp * β tp → (α → β) tp
```

The reification function now has type α `tp` $\to$ α `vl` $\to$ α `nf`: given a deep type `tp` whose corresponding shallow type is α, and a value of type α `vl`, `reify` yields a normal form of type α `nf`:

```
let rec reify : type a. a tp → a vl → a nf = fun a v → match a, v with
  | Arr (a, b), VFun f → NLam (fun x → reify b (f (reflect a (AVar x))))
  | Base, VBase v → let (Atom r) = v in NAt r

and reflect : type a. a tp → a at → a vl = fun a r → match a with
  | Arr (a, b) → VFun (fun x → reflect b (AApp (r, reify a x)))
  | Base → VBase (Atom r)
```

As in Section 3.1, because the match between types and terms is ensured statically, there is no need for any exception as in Section 2.2. Otherwise, the code is the same.

Let us now address the definition of `base`. As before, its values should contain atoms: at base type, terms are interpreted by atoms [36]. But one question remains: what is the type of atoms in the interpretation of the base type? Let us call this type X and let us rely on

the implementation as a guideline. In the base case of `reflect`, the type of `r` is refined to `base at`, and the expected type is `base`. Since Atom makes a `base` from an X `at`, we must have $X =$ `base`. Similarly in the base case of `reify`, the type of `v` is `base`, so `r` has type X `at`, NAt `r` has type X `nf`. Since the awaited type is `base nf`, we must have $X =$ `base`. The definition of type `base` is thus:

```
and base = Atom of base at
```

This type has no (normalizing) closed inhabitants: they are only constructed and deconstructed during reification and reflection. Its definition is faithful to previous formalizations, where the interpretation of the base type is the set of atomic terms at base type.

Finally, composing evaluation and reification, we obtain a typeful NbE function that is guaranteed to map well-typed terms to well-typed normal forms of the same type:

```
let nbe : type a. a tp → a tm → a nf = fun a m → reify a (eval m)
```

This function can be read as a cut elimination for intuitionistic logic, apart from termination which is not ensured by OCaml, but is a meta-argument: all three functions `eval`, `reify` and `reflect` are defined by structural induction over their first argument.

3.3 Application: `printf`, revisited

This section presents an application combining ideas from above: the offline specialization of `printf` with respect to a formatting directive, using NbE as a partial-evaluation engine. Given the same formatting directive as in Section 2.3, the program

```
fun x y z t → printf ex_directive x y z t
```

is specialized into the normal form

```
fun x y z t → string_of_int x ^ "␣*␣" ^ y ^ "␣=␣" ^ string_of_int z ^ "␣in␣" ^ t
```

in which `ex_directive` has been inlined and part of its processing has been carried out. This specialization is guaranteed to preserve types.

In Section 2.3, `kprintf` was mapping directives to the standard domain of OCaml primitive types. The idea here is to replace the primitive functions (concatenation `(^)`, `string_of_int`, `string_of_string`) by a non-standard, syntactic model. By reifying the evaluated program, we obtain a residual term in normal form.

First, we enlarge our representation of atoms (the type α `at`) with these primitive functions and uninterpreted objects of the types involved (to allow values of different types, we index the type `base` with a type variable, without consequence on its definition):

```
and α at = (* ... *)
  | APrim : α → α base at
  | AConcat : string base at * string base at → string base at
  | AStringOfInt : int base at → string base at
```

Since we strictly extended the definition of atoms and `reify` and `reflect` do not match on them, we can reuse these two functions from Section 3.2 as they are.

The primitive functions can now be interpreted as their residual expressions, atoms, instead of as their standard meanings:

```
type int_ = int base at
type string_ = string base at
let string_of_string i = APrim i
```

```
let string_of_int x = AStringOfInt x
let (^) s t = AConcat (s, t)
```

The non-standard `printf` is the result of pasting the code from Section 2.3 at this point, replacing types `int` and `string` by `int_` and `string_`, respectively.

▶ **Example 2.** Let us take this non-standard `printf` function, apply it to our example formatting directive and reify the result at the type of the function:

```
let residual =
  let box f = VFun (fun (VBase (Atom r)) → f r) in
  reify (Arr (Base, Arr (Base, Arr (Base, (Arr (Base, Base))))))
    (box (fun x → box (fun y → box (fun z → box (fun t →
           reflect Base (printf ex_directive x y z t))))))
```

We obtain the specialized program building the final string: `residual` is the normal form mentioned above (this can ben witnessed by pretty-printing it, or converting it to a de Bruijn representation [7]).

▶ Remark. NbE is type-directed, which leads to a completely offline partial evaluator: there is no need to explicitly check at each step of the program whether its result is statically known or not. It differs in that sense from the online partial evaluator proposed by Carette *et al.* [18]. Note that we could nonetheless perform online simplifications in our non-standard primitive functions [26].

4 Typeful Normalization by Evaluation in CPS

In Section 3.1, we defined an evaluation function for our object language. It is concise, but leaves no choice of evaluation order or definable control structures: they are inherited from the programming language of discourse, OCaml. In particular, it does not scale seamlessly for disjoint sums and not at all for call/cc:

sums: There is no simple notion of unique normal form for the λ-calculus with sums because of commuting conversions [43]. NbE with sums was nevertheless developed with delimited control operators [24, 34, 43] and constrained representations of unique normal forms were developed as well [2, 9]. Here, we bypass delimited control operators by writing the evaluation function in CPS, and we accept that normal forms are defined modulo commuting conversions (our notion of η-expansion is thus limited by them).

call/cc: Now that the evaluation function is written in CPS, it is simple to handle call/cc, and the resulting normalization function can immediately be used for programs extracted from classical proofs [29, 50].[5]

In this section, we show how to define *typeful* CPS evaluation and reification for the simply-typed λ-calculus with Boolean conditionals and call/cc. Our continuation-passing evaluation function maps source terms to continuation-passing values that await a continuation, and allows us to choose the evaluation order and to extend our source language. As in Section 3.2, we can then reify these continuation-passing values to a dedicated syntax of normal forms in CPS.

We present the formalization in call by value first (Sections 4.1 to 4.3), and then just sketch the call-by-name variant (Section 4.4).

[5] Another choice could have been shift and reset, as Ilik did in Coq [44].

4.1 Typing CPS values

When evaluating in CPS a term of type A, it is well-known [49] that its denotation is typed by the CPS-transformed type $\lceil A \rceil$, defined by:

$$\lceil A \rceil = (\lfloor A \rfloor \to o) \to o \qquad \lfloor p \rfloor = p$$
$$\lfloor A \to B \rfloor = \lfloor A \rfloor \to \lceil B \rceil \qquad \lfloor \mathsf{bool} \rfloor = \mathsf{bool}$$

where p is an (uninstantiated) base type, o is the type of answers, and bool is the type of Booleans. The call-by-value transformation can be reflected in the GADT that encodes CPS-values:

```
type α vl =
  | VFun : (α vl → β md) → (α → β) vl
  | VBase : base → base vl
  | VBool : bool → bool vl
and α md = (α vl → o) → o
```

The type `o` of answers is left unspecified for the moment. Note that the codomain of a function of type `(α → β) vl` expects a continuation (i.e., has type `β md`). For instance, the CPS-transformed applicator is written as follows:

```
let app : type a b. ((a → b) → a → b) vl =
  VFun (fun (VFun f) k → k (VFun (fun x k → f x (fun v → k v))))
```

4.2 Evaluation

Let us now extend the syntax of terms with an if statement and with call/cc:

```
type α tm = (* ... *)
  | If : bool tm * α tm * α tm → α tm
  | CC : ((α → β) → α) tm → α tm
```

Their typing is standard; call/cc has the type of Peirce's law [39]. Values of type bool are encoded as, e.g., `Var (VBool true)` (remember that `α x = α vl`).

Now, function `eval` directly maps an `α tm` to an `α md`. Its code can be obtained by CPS-transforming `eval` in Section 3.1 with the extra cases:

```
let rec eval : type a. a tm → a md = function
  | Var x → fun c → c x
  | Lam f → fun c → c (VFun (fun x k → eval (f x) k))
  | App (m, n) → fun c → eval m (fun (VFun f) → eval n (fun n → f n c))
  | If (b, m, n) → fun c → eval b (fun (VBool b) →
      if b then eval m c else eval n c)
  | CC m → fun c → eval m (fun (VFun f) → f (VFun (fun x k → c x)) c)
```

The if case is of no surprise, and could as well have been defined in direct style. The call/cc case captures the continuation `c` into a function, as customary in Scheme.

4.3 Reification

Now that the domain of `reify`, i.e., the values `α vl`, is in the image of the CPS transformation, we can CPS-transform the reification function of Section 3.2 as well. The types of `reify`

and `reflect` will thus be respectively α `tp` $\to$ α `vl` $\to$ (α `nf` $\to$ `o`) $\to$ `o` and α `tp` $\to$ α `at` $\to$ (α `vl` $\to$ `o`) $\to$ `o`. Consequently, the constructor NLam now takes a CPS-transformed function of type α `y` $\to$ β `k` $\to$ `o`, where α `k` = α `v` $\to$ `o` and α `v` = α `nf`.

Because of the latter function space, this data type is not a proper weak HOAS. But we can leave types α `k` and α `v` abstract – call these respectively continuation and value variables (α `y` is the domain of source variables):

```
type α k and α v and α y
```

We treat the answer type `o` algebraically, i.e., we instantiate it by all the operations involving continuation and value variables. There are two of them: applying an α `k` to a normal form in `reify` – call it SRet, and binding a value to an application in `reflect` – call it SBind (previous applications just become value nodes AVal). We are left with the type declarations:

```
and o =
  | SRet : α k * α nf → o
  | SBind : (α → β) at * α nf * (β v → o) → o
and α nf =
  | NLam : (α y → β k → o) → (α → β) nf
  | NAt : base at → base nf
and α at =
  | AVar of α y
  | AVal of α v
```

This typed syntax is in weak HOAS since the domains of variables are abstract. It has in fact been used since the late 1990's [10] to characterize normal forms in CPS: terms of type `o` are traditionally called *serious terms* after Reynolds [52], and represent computations. Note that they do not carry a type like α `nf` and α `at` since they form the type of answers; instead, its constructors act as existentials, linking together types of normal forms, variables and atoms, and hiding them away. Normal forms are traditionally called *trivial terms*, again after Reynolds [52].

Before displaying the code, let us extend the development to Booleans. First, we add the extra case to the type α `tp`:

```
type α tp = (* ... *)
  | Bool : bool tp
```

Then, we add Booleans and conditional expressions to normal forms and serious terms, respectively:

```
and o = (* ... *)
  | SIf : bool at * o * o → o
and α nf = (* ... *)
  | NBool : bool → bool nf
```

At last, the full definition of `reify` and `reflect` with Booleans reads:

```
let rec reify : type a. a tp → a vl → (a nf → o) → o =
  fun a v → match a, v with
    | Arr (a, b), VFun f → fun c → c (NLam (fun x k →
        reflect a (AVar x) (fun x → f x (fun v →
            reify b v (fun v → SRet (k, v))))))
    | Base, VBase (Atom r) → fun c → c (NAt r)
    | Bool, VBool b → fun c → c (NBool b)
```

```
and reflect : type a. a tp → a at → (a vl → o) → o =
  fun a x → match a, x with
    | Arr (a, b), f → fun c → c (VFun (fun x k →
        reify a x (fun x → SBind (f, x, fun v →
            reflect b (AVal v) (fun v → k v)))))
    | Base, r → fun c → c (VBase (Atom r))
    | Bool, b → fun c → SIf (b, c (VBool true), c (VBool false))
```

Similarly to the direct-style version, these two functions can be seen as performing a two-level η-expansion, this time with the expansion rules of CPS with sums [31]. This fact dictates the treatment of conditionals in the last line: they are serious terms, and duplicate the context `c` in their two branches.

We can now compose evaluation and reification to obtain normalization. A CPS value is reified as a program in normal form: a serious term abstracted by its initial continuation. NbE in CPS thus returns such an abstraction:

```
type α c = Init of (α k → o)
let nbe : type a. a tp → a tm → a c = fun a m →
  Init (fun k → eval m (fun m → reify a m (fun v → SRet (k, v))))
```

As an epilogue, we strip out the resulting syntax of its type annotations to obtain the familiar syntax of call-by-value CPS normal forms:

$$
\begin{array}{ll}
P ::= \lambda k.\, S & \text{Programs} \\
S ::= k\, T \mid R\, S\, (\lambda v.\, S) \mid \mathsf{if}(R, S, S) & \text{Serious terms} \\
T ::= \lambda y k.\, S \mid \mathsf{true} \mid \mathsf{false} \mid R & \text{Trivial terms} \\
R ::= y \mid v & \text{Atoms}
\end{array}
$$

As in the direct-style case, it is syntactically impossible to form a redex in this syntax, thanks to the stratification of trivial terms and atoms.

4.4 In call by name

In call by name, the domains of functions are also computations (i.e., expecting a continuation), as presented in Section 4.1. This transformation is reflected:

- in the type of values in that functions now expect a continuation:

```
type α vl = (* ... *)
  | VFun : (α md → β md) → (α → β) vl
```

- in the variables of the source language that now range over thunks instead of values:

```
and α x = α md
```

- and in the variables of the target language: they are now serious terms, and are associated with a continuation binding their values; for the same reason, the argument to a "bind" is now a thunk:

```
and o =
  | SRet : α k * α nf → o
  | SBind : (α → β) at * (α k → o) * (β v → o) → o
  | SIf : bool at * o * o → o
  | SVar : α y * (α v → o) → o
and α nf =
```

```
  | NLam : (α y → β k → o) → (α → β) nf
  | NBool : bool → bool nf
  | NAt : base at → base nf
and α at =
  | AVal : α v → α at
```

Evaluation and reification functions are modified *mutatis mutandis*:

```
let rec eval : type a. a tm → a md = function
  | Var x → fun c → x c
  | Lam f → fun c → c (VFun (fun x k → eval (f x) k))
  | App (m, n) → fun c → eval m (fun (VFun f) → f (eval n) c)
  | Bool b → fun c → c (VBool b)
  | If (b, m, n) → fun c → eval b
      (function VBool true → eval m c | VBool false → eval n c)
  | CC m → fun c → eval m (fun (VFun f) →
      f (fun k → k (VFun (fun x k → x c))) c)
```

```
let rec reify : type a. a tp → a vl → (a nf → o) → o =
  fun a v → match a, v with
  | Arr (a, b), VFun f → fun c → c (NLam (fun y k →
        f (fun k → SVar (y, fun v → reflect a (AVal v) k))
          (fun v → reify b v (fun v → SRet (k, v)))))
  | Bool, VBool b → fun c → c (NBool b)
  | Base, VBase (Atom r) → fun c → c (NAt r)

and reflect : type a. a tp → a at → (a vl → o) → o =
  fun a x → match a, x with
  | Arr (a, b), f → fun c → c (VFun (fun x k →
      SBind (f, (fun k → x (fun v → reify a v (fun v → SRet (k, v)))),
             (fun v → reflect b (AVal v) k))))
  | Bool, b → fun c → SIf (b, c (VBool true), c (VBool false))
  | Base, r → fun c → c (VBase (Atom r))
```

As before, these two functions can be seen as performing a two-level η-expansion, this time with the expansion rules of call-by-name CPS [41].

We can finally compose evaluation and reification to obtain normalization. As in the call-by-value case, NbE in call-by-name CPS returns a program, i.e., a serious term abstracted by the initial continuation:

```
let nbe : type a. a tp → a tm → (a nf → o) → o =
  fun a m k → eval m (fun m → reify a m k)
```

As an epilogue, we strip out the resulting syntax of its type annotations to obtain the familiar syntax of call-by-name CPS normal forms:

$$
\begin{array}{lr}
P ::= \lambda k.\, S & \text{Programs} \\
S ::= k\,T \mid R\,(\lambda k.\, S)\,(\lambda v.\, S) \mid \mathsf{if}(R, S, S) \mid y\,(\lambda v.S) & \text{Serious terms} \\
T ::= \lambda y k.\, S \mid \mathsf{true} \mid \mathsf{false} \mid R & \text{Trivial terms} \\
R ::= v & \text{Atoms}
\end{array}
$$

Again, it is syntactically impossible to form a redex in this syntax.

5 Summary and Future Work

We have presented the first typeful implementation of NbE for the simply-typed λ-calculus in the minimalistic setting of a general-purpose programming language with GADTs. To the best of our knowledge, our implementation is the first one to ensure by typing that its output is not only in β-normal form, but also in η-long form. We have illustrated how NbE achieves partial evaluation by specializing a typeful version of `printf` with respect to any given formatting directive. By CPS-transforming our typeful implementation, we have obtained systematically the syntax and typing rules of normal forms in CPS. Finally, we have presented the first typeful implementation of NbE for the simply-typed λ-calculus with sums and control operators in the same minimalistic setting. This normalization function can be used for programs extracted from classical proofs, and the resulting normal form can then be mapped back to direct style [23, 29].

Future work includes developing a version of NbE that is parameterized by an arbitrary monad (i.e., not just the identity monad or a continuation monad). In this version, the non-standard evaluation function is monadic. Monadic reification with effect preservation seems like a tall order, but given a monad, reification towards a (well-typed but non-monadic) normal form seems in sight: it could be achieved using the type transformation associated to this given monad; a monadic version of the direct-style transformation would then be necessary to map this non-monadic normal form to a monadic normal form. Such a monadic version of NbE would make it possible to normalize programs whose effects can be described with monads, e.g., probabilistic or stateful computations.

Acknowledgments. This work was carried out while the two last authors were visiting the first, in the fall of 2013. We are grateful to our anonymous reviewers for their helpful comments.

References

1 Andreas Abel, Thierry Coquand, and Peter Dybjer. Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In Jerzy Marcinkowski, editor, *Proceedings of the 22nd IEEE Symposium on Logic in Computer Science (LICS 2007)*, pages 3–12, Wroclaw, Poland, July 2007. IEEE Computer Society Press.

2 Thorsten Altenkirch, Peter Dybjer, Martin Hofmann, and Philip J. Scott. Normalization by evaluation for typed lambda calculus with coproducts. In Joseph Halpern, editor, *Proceedings of the Sixteenth Annual IEEE Symposium on Logic in Computer Science*, pages 203–210, Boston, Massachusetts, June 2001. IEEE Computer Society Press.

3 Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction-free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter Johnstone, editors, *Category Theory and Computer Science*, number 953 in Lecture Notes in Computer Science, pages 182–199, Cambridge, UK, August 1995. Springer-Verlag.

4 Michaël Armand, Germain Faure, Benjamin Grégoire, Chantal Keller, Laurent Théry, and Benjamin Werner. A modular integration of SAT/SMT solvers to Coq through proof witnesses. In Jouannaud and Shao [45], pages 135–150.

5 Kenichi Asai. On typing delimited continuations: three new solutions to the printf problem. *Higher-Order and Symbolic Computation*, 22(3):275–291, 2009.

6 Kenichi Asai, Luminous Fennell, Peter Thiemann, and Yang Zhang. A type theoretic specification for partial evaluation. In Olaf Chitil, Andy King, and Olivier Danvy, editors, *Proceedings of the 16th ACM SIGPLAN International Conference on Principles and*

Practice of Declarative Programming (PPDP'14), pages 57–68, Canterbury, UK, September 2014. ACM Press.

7 Robert Atkey, Sam Lindley, and Jeremy Yallop. Unembedding domain-specific languages. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 37–48. ACM, 2009.

8 Vincent Balat. *Une étude des sommes fortes: isomorphismes et formes normales*. PhD thesis, PPS, Université Denis Diderot (Paris VII), Paris, France, December 2002.

9 Vincent Balat, Roberto Di Cosmo, and Marcelo P. Fiore. Extensional normalisation and type-directed partial evaluation for typed lambda calculus with sums. In Xavier Leroy, editor, *Proceedings of the Thirty-First Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 39, No. 1, pages 64–76, Venice, Italy, January 2004. ACM Press.

10 Vincent Balat and Olivier Danvy. Strong normalization by type-directed partial evaluation and run-time code generation. In Xavier Leroy and Atsushi Ohori, editors, *Proceedings of the Second International Workshop on Types in Compilation*, number 1473 in Lecture Notes in Computer Science, pages 240–252, Kyoto, Japan, March 1998. Springer-Verlag.

11 Freiric Barral. *Decidability for non standard conversions in typed λ-calculus*. PhD thesis, Ludvig-Maximilians-Universität and Université Paul Sabatier, München, Germany and Toulouse, France, June 2008.

12 Ulrich Berger. Program extraction from normalization proofs. In Marc Bezem and Jan Friso Groote, editors, *Typed Lambda Calculi and Applications*, number 664 in Lecture Notes in Computer Science, pages 91–106, Utrecht, The Netherlands, March 1993. Springer-Verlag.

13 Ulrich Berger, Stefan Berghofer, Pierre Letouzey, and Helmut Schwichtenberg. Program extraction from normalization proofs. *Studia Logica*, 82(1):25–49, 2006.

14 Ulrich Berger, Matthias Eberl, and Helmut Schwichtenberg. Normalization by evaluation. In Bernhard Möller and John V. Tucker, editors, *Prospects for hardware foundations (NADA)*, number 1546 in Lecture Notes in Computer Science, pages 117–137, Berlin, Germany, 1998. Springer-Verlag.

15 Ulrich Berger and Helmut Schwichtenberg. An inverse of the evaluation functional for typed λ-calculus. In Gilles Kahn, editor, *Proceedings of the Sixth Annual IEEE Symposium on Logic in Computer Science*, pages 203–211, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.

16 Mathieu Boespflug. *Conception d'un noyau de vérification de preuves pour le λΠ-calcul modulo*. PhD thesis, École Polytechnique, Palaiseau, France, January 2011.

17 Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jouannaud and Shao [45], pages 362–377.

18 Jacques Carette, Oleg Kiselyov, and Chung-chieh Shan. Finally tagless, partially evaluated: Tagless staged interpreters for simpler typed languages. *Journal of Functional Programming*, 19(5):509–543, 2009.

19 James Cheney and Ralf Hinze. First-class phantom types. Technical Report 1901, Computing and Information Science, Cornell University, Ithaca, New York, 2003.

20 Adam Chlipala. Parametric higher-order abstract syntax for mechanized semantics. In James Hook and Peter Thiemann, editors, *Proceedings of the 2008 ACM SIGPLAN International Conference on Functional Programming (ICFP'08)*, SIGPLAN Notices, Vol. 43, No. 9, pages 143–156, Victoria, British Columbia, September 2008. ACM Press.

21 Thierry Coquand and Peter Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, 7:75–94, 1997.

22 Djordje Čubrić, Peter Dybjer, and Philip J. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, 8:153–192, 1998.

23 Olivier Danvy. Back to direct style. *Science of Computer Programming*, 22(3):183–195, 1994.

24 Olivier Danvy. Type-directed partial evaluation. In Guy L. Steele Jr., editor, *Proceedings of the Twenty-Third Annual ACM Symposium on Principles of Programming Languages*, pages 242–257, St. Petersburg Beach, Florida, January 1996. ACM Press.

25 Olivier Danvy. Functional unparsing. *Journal of Functional Programming*, 8(6):621–625, 1998.

26 Olivier Danvy. Online type-directed partial evaluation. In Masahiko Sato and Yoshihito Toyama, editors, *Proceedings of the Third Fuji International Symposium on Functional and Logic Programming*, pages 271–295, Kyoto, Japan, April 1998. World Scientific.

27 Olivier Danvy and Peter Dybjer, editors. *Proceedings of the 1998 APPSEM Workshop on Normalization by Evaluation (NBE 1998)*, BRICS Note Series NS-98-8, Gothenburg, Sweden, May 1998. BRICS, Department of Computer Science, Aarhus University. Available online at `http://www.brics.dk/~nbe98/programme.html`.

28 Olivier Danvy and Andrzej Filinski. Abstracting control. In Mitchell Wand, editor, *Proceedings of the 1990 ACM Conference on Lisp and Functional Programming*, pages 151–160, Nice, France, June 1990. ACM Press.

29 Olivier Danvy and Julia L. Lawall. Back to direct style II: First-class continuations. In William Clinger, editor, *Proceedings of the 1992 ACM Conference on Lisp and Functional Programming*, LISP Pointers, Vol. V, No. 1, pages 299–310, San Francisco, California, June 1992. ACM Press.

30 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. The essence of eta-expansion in partial evaluation. *Lisp and Symbolic Computation*, 8(3):209–227, 1995.

31 Olivier Danvy, Karoline Malmkjær, and Jens Palsberg. Eta-expansion does The Trick. *ACM Transactions on Programming Languages and Systems*, 8(6):730–751, 1996.

32 Olivier Danvy, Morten Rhiger, and Kristoffer Rose. Normalization by evaluation with typed abstract syntax. *Journal of Functional Programming*, 11(6):673–680, 2001.

33 Peter Dybjer and Andrzej Filinski. Normalization and partial evaluation. In Gilles Barthe, Peter Dybjer, Luís Pinto, and João Saraiva, editors, *Applied Semantics – Advanced Lectures*, number 2395 in Lecture Notes in Computer Science, pages 137–192, Caminha, Portugal, September 2000. Springer-Verlag.

34 Andrzej Filinski. Normalization by evaluation for the computational lambda-calculus. In Samson Abramsky, editor, *Typed Lambda Calculi and Applications, 5th International Conference, TLCA 2001*, number 2044 in Lecture Notes in Computer Science, pages 151–165, Kraków, Poland, May 2001. Springer.

35 Andrew Gacek, Dale Miller, and Gopalan Nadathur. A two-level logic approach to reasoning about computations. *Journal of Automated Reasoning*, 49(2):241–273, 2012.

36 François Garillot and Benjamin Werner. Simple types in type theory: Deep and shallow encodings. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics, 20th International Conference, TPHOLs 2007*, number 4732 in Lecture Notes in Computer Science, pages 368–382, Kaiserslautern, Germany, September 2007. Springer.

37 Jacques Garrigue and Didier Rémy. Ambivalent types for principal type inference with gadts. In Chung-chieh Shan, editor, *Programming Languages and Systems – 11th Asian Symposium, APLAS 2013, Melbourne, VIC, Australia, December 9-11, 2013. Proceedings*, volume 8301 of *Lecture Notes in Computer Science*, pages 257–272. Springer, 2013.

38 George Gonthier. Formal proof – the four-color theorem. *Notices of the AMS*, 55(11):1382–1393, December 2008.

39 Timothy G. Griffin. A formulae-as-types notion of control. In Paul Hudak, editor, *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 47–58, San Francisco, California, January 1990. ACM Press.

40 Peter Hancock. The model of computable terms. In Danvy and Dybjer [27]. Available online at `http://www.brics.dk/~nbe98/programme.html`.

41 John Hatcliff and Olivier Danvy. Thunks and the λ-calculus. *Journal of Functional Programming*, 7(3):303–319, 1997.

42 Noriko Hirota and Kenichi Asai. Formalizing a correctness property of a type-directed partial evaluator. In Nils Anders Danielsson and Bart Jacobs, editors, *Proceedings of the 2014 ACM SIGPLAN Workshop on Programming Languages meets Program Verification, PLPV 2014, January 21, 2014, San Diego, California, USA, Co-located with POPL'14*, pages 41–46. ACM, 2014.

43 Danko Ilik. *Constructive Completeness Proofs and Delimited Control*. PhD thesis, École Polytechnique, Palaiseau, France, October 2010.

44 Danko Ilik. A formalized type-directed partial evaluator for shift and reset. In Ugo de'Liguoro and Alexis Saurin, editors, *Proceedings of the First Workshop on Control Operators and their Semantics, COS 2013, Eindhoven, The Netherlands, June 24-25, 2013*, volume 127 of *EPTCS*, pages 86–100, 2013.

45 Jean-Pierre Jouannaud and Zhong Shao, editors. *Certified Programs and Proofs – First International Conference, CPP 2011*, number 7086 in Lecture Notes in Computer Science, Kenting, Taiwan, December 2011. Springer.

46 Oleg Kiselyov. Type-safe functional formatted IO, 2008. Web post, available at `http://okmij.org/ftp/typed-formatting/`.

47 Oleg Kiselyov. Typed tagless final interpreters. In Jeremy Gibbons, editor, *Generic and Indexed Programming – International Spring School, SSGIP 2010, Oxford, UK, March 22-26, 2010, Revised Lectures*, volume 7470 of *Lecture Notes in Computer Science*, pages 130–174. Springer, 2010.

48 Per Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In *Proceedings of the Third Scandinavian Logic Symposium (1972)*, volume 82 of *Studies in Logic and the Foundation of Mathematics*, pages 81–109. North-Holland, 1975.

49 Albert R. Meyer and Mitchell Wand. Continuation semantics in typed lambda-calculi (summary). In Rohit Parikh, editor, *Logics of Programs – Proceedings*, number 193 in Lecture Notes in Computer Science, pages 219–224, Brooklyn, New York, June 1985. Springer.

50 Chetan R. Murthy. *Extracting Constructive Content from Classical Proofs*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, 1990.

51 Matthias Puech. Parametric HOAS with first-class modules. `https://syntaxexclamation.wordpress.com/2014/06/27/parametric-hoas-with-first-class-modules/`, 2014.

52 John C. Reynolds. Definitional interpreters for higher-order programming languages. In *Proceedings of 25th ACM National Conference*, pages 717–740, Boston, Massachusetts, 1972.

53 John C. Reynolds. Normalization and functor categories. In Danvy and Dybjer [27]. Available online at `http://www.brics.dk/~nbe98/programme.html`.

54 Hongwei Xi, Chiyan Chen, and Gang Chen. Guarded recursive datatype constructors. In Greg Morrisett, editor, *Proceedings of the Thirtieth Annual ACM Symposium on Principles of Programming Languages*, SIGPLAN Notices, Vol. 38, No. 1, pages 224–235, New Orleans, Louisiana, January 2003. ACM Press.

55 Jeremy Yallop and Oleg Kiselyov. First-class modules: hidden power and tantalizing promises. Presented at the 2010 Workshop on ML, 2010.

56 Zhe Yang. *Language Support for Program Generation: Reasoning, Implementation, and Applications*. PhD thesis, Computer Science Department, New York University, New York, New York, August 2001.

Dialectica Categories and Games with Bidding

Jules Hedges

Queen Mary University of London, UK
j.hedges@qmul.ac.uk

Abstract

This paper presents a construction which transforms categorical models of additive-free propositional linear logic, closely based on de Paiva's dialectica categories and Oliva's functional interpretations of classical linear logic. The construction is defined using dependent type theory, which proves to be a useful tool for reasoning about dialectica categories. Abstractly, we have a closure operator on the class of models: it preserves soundness and completeness and has a monad-like structure. When applied to categories of games we obtain 'games with bidding', which are hybrids of dialectica and game models, and we prove completeness theorems for two specific such models.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Linear logic, Dialectica categories, categorical semantics, model theory, game semantics, dependent types, functional interpretations

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.89

1 Introduction

This paper presents a construction which transforms categorical models of additive-free propositional linear logic, closely based on de Paiva's dialectica categories and Oliva's functional interpretations of classical linear logic.

The dialectica categories [9] are a family of models of intuitionistic logic, and classical and intuitionistic linear logic, based on Gödel's dialectica interpretation. Dialectica models of classical linear logic are described in [9], based on earlier models of intuitionistic logic and intuitionistic linear logic in [7]. Historically they were the first models of linear logic to not equate multiplicative and additive units, and they have been generalised in several ways, for example [13] defines dialectica categories starting only from a partially ordered fibration. The construction in this paper is closely related to [8] and [11]; the similarities and differences between that construction and the original dialectica categories is discussed in those papers. While most of the literature on dialectica categories aims to construct large classes of structured categories and then characterise those which are sound models of some logic, the aim of this paper is rather different: to construct a small number of concrete models which can be interpreted as game models and are amenable to a proof-theoretic analysis of the valid formulas, and in particular are as close as possible to being complete models of linear logic.

Based on de Paiva's models, [28] gave a syntactic dialectica and Diller-Nahm interpretation to first order affine logic, and [24] to classical linear logic. The semantics of the Diller-Nahm variant is explored in detail in chapter 4 of [9], and will be used in this paper. A completeness theorem is given in [25] for the dialectica interpretation, based on Gödel's original completeness theorem for Heyting arithmetic [1], which has not been exploited so far in the semantic literature. This relies on a small but crucial modification to de Paiva's interpretation of the linear exponentials. The Diller-Nahm interpretation of linear logic appears in [24] and [26],

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 89–110

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

although no completeness proof for the Diller-Nahm interpretation of linear logic appears in the literature, to the author's knowledge.

The dialectica interpretation, intuitively, is a proof translation which takes a formula φ to a quantifier-free formula $|\varphi|^x_y$ in which the variables x and y appear free. The variable x represents 'witnesses', or evidence that a theorem is true, and y represents 'counter-witnesses', or evidence that a theorem is false. The validity of a theorem is then reduced to the existence of a witness which defeats every counter-witness, that is, $\exists x \forall y. |\varphi|^x_y$. However even if φ is a first-order formula the variables x and y may have higher types. The original purpose was to prove the relative consistency of Heyting arithmetic to the quantifier-free language called system T, however the dialectica interpretation is now mainly used to give a computational interpretation to theorems of classical analysis, see [17].

The semantic equivalent to the dialectica interpretation, at least from the point of view of this paper, is to replace the formula $|\varphi|^x_y$ with a double-indexed family of objects in some model $\mathcal{R}$. We can imagine that we are composing the syntactic proof translation with a semantic interpretation of formulas. The fact that the dialectica interpretations of linear negation and multiplicative conjunction are given recursively by

$$\left|\varphi^\perp\right|^y_x = \left(|\varphi|^x_y\right)^\perp$$

$$|\varphi \otimes \psi|^{x,u}_{f,g} = |\varphi|^x_{fu} \otimes |\psi|^u_{gx}$$

(in particular, that the same connectives occur on the right hand side) tells us that $\mathcal{R}$ must have a sound interpretation of these connectives. This leads us to the construction in [8], which builds a dialectica category from a posetal model of multiplicative linear logic, or *lineale* [10]. The dialectica interpretation eliminates additives (in the sense that additives do not appear on the right hand side of the corresponding formulas), and it is also possible to eliminate exponentials in a sound way by defining

$$|!\varphi|^x_f = |\varphi|^x_{fx}$$

However the completeness theorem of [25] relies on changing this definition to

$$|!\varphi|^x_f = !\,|\varphi|^x_{fx}$$

To interpret this semantically $\mathcal{R}$ must also have a sound interpretation of the exponential, which leads to our construction of dialectica categories beginning from an arbitrary model of multiplicative-exponential linear logic (**MELL**). Thus this work can be seen as the result of a 'dialogue' between syntax and semantics.

Overall, we have a construction $\mathfrak{D}$ which takes a model of **MELL** to a model of **LL**. The first of two aims of this paper is to explore the abstract properties of $\mathfrak{D}$. We prove in section 6 that $\mathfrak{D}$ is functorial, and that it has a monad-like structure on a particular category of models of **MLL**, although one of the monad laws fails and even the weaker result fails to extend to **MELL**. This is closely related to the main theorem in [12]. (We could also explore the 2-categorical properties of $\mathfrak{D}$, but that is left for later work.) We also prove that $\mathfrak{D}$ preserves soundness (section 3) and completeness (section 5) for **MELL**, so we can justify calling it a 'closure operator' on models.

The second aim of this paper is to construct specific dialectica categories (rather than axiomatically-defined families) which have logical completeness properties. This requires that the underlying model also has completeness properties, which in practice means constructing a dialectica category from a category of games. In section 4 we informally describe such a

dialectica category as a category of 'games with bidding', greatly extending the comments in [6] on viewing dialectica categories as game models. In particular in section 5 we consider 'Hyland-Ong games with bidding' based on [15], and 'asynchronous games with bidding' based on [20], and prove that these models are complete respectively for **MLL** and **MELL**.

The model of asynchronous games with bidding, in particular, is an extremely interesting model because the starting model has the strongest possible completeness theorem, namely it is fully complete for **MELL**. An analysis of the formulas containing additives which are valid in this model will be carried out in a follow-up paper, but an overview of the argument is given in section 7.

There are two main technical ideas in this paper which contribute to our two aims. The first is that we replace the posets of [8] and [16] with categories, and use dependent type theory in defining and reasoning about our models. If our metatheory has choice this formally gains nothing, however in practice dependent type theory proves to be a powerful tool. Dialectica categories were used with dependent types in [5], but in a semantic rather than a syntactic way. Our use of syntactic dependent types will be justified in particular in sections 5 and 6, which would be hard to formalise otherwise. It also suggests the implementation of this construction (and the formalisation of the proofs in this paper) in a dependently typed programming language. This would require libraries for 2-category theory and monoidal category theory, and would be an interesting way to embed linear reasoning into a proof assistant. This was carried out in Coq in [3], for the special case in which the underlying model is the built-in type of propositions.

The second idea is that we work with the linear-nonlinear semantics of **MELL** and **LL** given in [4]. Although the relationship between linear categories and linear-nonlinear adjunctions is well understood, a direct formulation of dialectica categories as linear-nonlinear adjunctions is still quite informative: it allows the relationship between the linear and intuitionistic dialectica categories to be clearly seen, and allows us to factor the exponential into four parts. This also suggests turning back around to syntax and studying a syntactic dialectica interpretation of linear-nonlinear logic.

Note that in this paper we are only considering *classical* linear logic. The differences between dialectica models of classical and intuitionistic linear logic are subtle: firstly for intuitionistic linear logic the sets of witnesses and counterexamples must *both* be nonempty, whereas for classical linear logic one may be empty; and secondly for intuitionistic linear logic we consider the bids in games with bidding to be sequential rather than simultaneous. Since the two logics coincide in the absence of additives, the difference will not often affect us.

2 The dialectica transformations of a category

In this section we will define the two dialectica transformations of a category, and relate them to the existing literature on dialectica categories. The game-semantic intuition corresponding to these definitions will be given in section 4.

Let R be an arbitrary category. We will define a category $\mathfrak{D}_l(R)$ called the *linear dialectica transformation* of R. The objects of $\mathfrak{D}_l(R)$ are double-indexed families $\mathcal{G}_Y^X$ where X and Y are arbitrary sets not both empty, and each $\mathcal{G}_y^x$ is an object of R. Throughout this paper we will specify such objects using the notation

$$\mathcal{G}_Y^X : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \cdots$$

where the right hand side is an expression in terms of x and y. Since X and Y will often be (dependent) pairs we will drop the parentheses, as is done in the proof theory literature.

Sometimes we will decorate witness and counter-witness variables with their individual types for clarity, as in

$$\mathcal{G}^{X\times U}_{Y\times V} : \begin{pmatrix} x : X, u : U \\ y : Y, v : V \end{pmatrix} \mapsto \cdots$$

A morphism from $\mathcal{G}^X_Y$ to $\mathcal{H}^U_V$ is an element of a dependent type in the category of sets:

$$\hom_{\mathfrak{D}_l(R)}\left(\mathcal{G}^X_Y, \mathcal{H}^U_V\right) = \sum_{\substack{f:X\to U \\ g:V\to Y}} \prod_{\substack{x:X \\ v:V}} \hom_R\left(\mathcal{G}^x_{gv}, \mathcal{H}^{fx}_v\right)$$

Hence a morphism is a triple (f, g, α) where $f : X \to U$, $g : V \to Y$ and α is a double-indexed family of R-morphisms

$$\alpha_{x,v} : \mathcal{G}^x_{gv} \to \mathcal{H}^{fx}_v$$

The proof-theoretic reading of this is that a morphism consists of a witness, together with a mapping that takes each counter-witness to a proof that the counter-witness is invalid. This is simply the type-theoretic interpretation of the usual dialectica interpretation of linear implication, with quantifiers replaced by dependent types.

For simplicity, in this paper we only explicitly use the set-theoretic interpretation of dependent type theory, however it should be straightforward to generalise to any model of dependent type theory. This would require R to be enriched over a locally cartesian closed category $\mathcal{C}$, and that we have a suitable fibration of objects of R over $\mathcal{C}$ to replace set-indexed families, similar to [13] (this idea was suggested in [14]).

In $\mathfrak{D}_l(R)$ the identity morphism on $\mathcal{G}^X_Y$ is given by the identity functions on X and Y together with identity morphisms in R. The composition of a morphism $\mathcal{G}^X_Y \multimap \mathcal{H}^U_V$ given by (f, g, α) and another $\mathcal{H}^U_V \multimap \mathcal{I}^P_Q$ given by (f', g', β) is given by $f' \circ f : X \to P$ and $g \circ g' : Q \to Y$, together with the composition

$$(\beta \circ \alpha)_{x,q} = \beta_{fx,v} \circ \alpha_{x,g'q} : \hom_R\left(\mathcal{G}^x_{g(g'q)}, \mathcal{I}^{f'(fx)}_q\right)$$

▶ **Lemma 1.** *Let R be any category, then $\mathfrak{D}_l(R)$ is a category with finite products and coproducts.*

Proof. By proposition 3.7 of [16]. ◀

Using the axiom of choice (at least in the case $\mathcal{C} = \mathbf{Set}$), this definition is equivalent to $M_N(\mathcal{C})$ in [8] where N is the posetal reflection of R (assuming a Grothendieck universe, since R will be large in general). To be clear, this definition is not intended to be exactly equivalent to the original dialectica categories in [9], which is more elegant and far more general but is hard to use for concrete calculations. In particular using type theory gives us explicit names for all of our morphisms, and this will make our life easier especially in sections 5 and 6. Moreover we can avoid using the axiom of choice in our metatheory, and so the contents of this paper could be directly implemented in a dependently typed programming language.

Next we will construct the Diller-Nahm translation $\mathfrak{D}_i(S)$ of an arbitrary category S with finite products. This construction is most closely related to that in [13], although we consider it in far less generality than in that paper. The objects of $\mathfrak{D}_i(S)$, as before, are double-indexed families $\mathcal{G}^X_Y$ where X and Y are sets not both empty and each $\mathcal{G}^x_y$ is an element of S. The hom-sets are defined by

$$\hom_{\mathfrak{D}_i(S)}\left(\mathcal{G}^X_Y, \mathcal{H}^U_V\right) = \sum_{\substack{f:X\to U \\ g:X\times V\to Y^*}} \prod_{\substack{x:X \\ v:V}} \hom_S\left(\prod_{y\in g(x,v)} \mathcal{G}^x_y, \mathcal{H}^{fx}_v\right)$$

Here Y^* is the set of finite multisets with elements in Y. This definition is the type-theoretic interpretation of the Diller-Nahm interpretation of intuitionistic implication

$$\exists f^{X\to U}, g^{X\times V\to Y^*} \forall x^X, v^V. \left(\forall y \in g(x,v). |\varphi|_y^x\right) \to |\psi|_v^{fx}$$

However we carefully distinguish 'internal' and 'external' quantifiers: the internal $\forall$ is interpreted as the categorical product in the underlying model, and the external $\exists\forall$ is interpreted as dependent types in $\mathcal{C}$.

In $\mathfrak{D}_i(S)$ the structure is very similar. If we have a morphism given by $f : X \to U$ and $g : X \times V \to Y^*$ and another given by $f' : U \to P$ and $g' : U \times Q \to V^*$ the composition is given by $f' \circ f : X \to P$ and

$$\lambda x^X, q^V. g'(fx, q) \gg\!= \lambda v^V. g(x, v) : X \times Q \to Y^*$$

together with composition in S. Here $\gg\!=$ is the bind operator of the finite multiset monad, where $l \gg\!= f$ applies f to each element of l, each giving a multiset, and collects the results with a union.

▶ **Lemma 2.** *Let S be any category with finite products, then $\mathfrak{D}_i(S)$ is a category with finite products.*

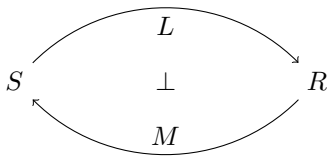
Proof. By section 3 of [13]. ◀

3 The dialectica transformation of a linear-nonlinear adjunction

We begin with a general definition of a model of **MELL** and a model of **LL**. A model of multiplicative linear logic (**MLL**) is given by a $*$-autonomous category R [2], that is, a symmetric monoidal closed category $(R, \otimes, \multimap, 1)$ with a functor $^\perp : R \to R$ and natural isomorphisms $^\perp \circ ^\perp \cong \mathrm{id}_R$ and

$$\hom_R(X \otimes Y, Z^\perp) \cong \hom_R(X, (Y \otimes Z)^\perp)$$

For the interpretation of exponentials we use the linear-nonlinear semantics of [4], which is surveyed in detail in [22]. A categorical model of **MELL** is given by a $*$-autonomous category R together with another category S with finite products and an adjunction



or, more briefly,

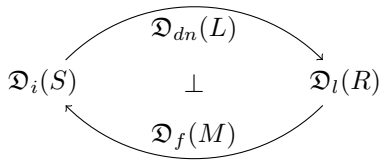
$$L \dashv M : R \to S$$

Here L (called *linearisation*) and M (called *multiplication*) are lax symmetric monoidal functors, that is, there are natural transformations

$$M(X) \times M(Y) \to M(X \otimes Y) \qquad \top \to M(1)$$
$$L(X) \otimes L(Y) \to L(X \times Y) \qquad 1 \to L(\top)$$

and the unit and counit of the adjunction must also respect the monoidal and cartesian monoidal structures (ie. the adjunction must be a *symmetric monoidal adjunction*). Such a setup is called a *linear-nonlinear adjunction*. Given this adjunction, the denotation of the exponential ! is the composition $L \circ M$, which is a comonad on R (and conversely, if we have a model in which ! is given explicitly we can recover S, M and L from the co-Kleisli adjunction). The entire model, which contains a pair of categories and functors and various natural transformations, will be denoted $\mathcal{R}$. For a model of **LL** we simply require that R also has finite products.

Given such a model of **MELL**, the dialectica transformation of this model will be a new pair of categories and a linear-nonlinear adjunction



The categories $\mathfrak{D}_l(R)$ and $\mathfrak{D}_i(S)$ are precisely the categories defined in the previous section. The transformations of the functors M and L will be given below. The transformed model as a whole will be denoted $\mathfrak{D}(\mathcal{R})$.

The interpretations of each connective in $\mathfrak{D}_l(R)$ is given in Figure 1.

▶ **Lemma 3.** *Let R be any $*$-autonomous category, then $\mathfrak{D}_l(R)$ is a $*$-autonomous category.*

Proof. By propositions 3.6 of [16]. ◀

Now we give the dialectica transformations $\mathfrak{D}_f(M)$ and $\mathfrak{D}_{dn}(L)$ of the multiplication and linearisation functors. The operation $\mathfrak{D}_f$ is a straightforward lifting operation. The subscript f stands for *functor* since this construction will be used in section 6 to give the action of $\mathfrak{D}$ on maps (or functors) of models. Suppose the multiplication functor is $M : R \to S$. The functor $\mathfrak{D}_f(M) : \mathfrak{D}_l(R) \to \mathfrak{D}_i(S)$ acts on objects $\mathcal{G}_Y^X$ of $\mathfrak{D}_l(R)$ by

$$(\mathfrak{D}_f(M)(\mathcal{G}))_Y^X : \binom{x}{y} \mapsto M(\mathcal{G}_y^x)$$

For the action of $\mathfrak{D}_f(M)$ on morphisms, suppose we have a morphism of $\mathfrak{D}_l(R)$ from $\mathcal{G}_Y^X$ to $\mathcal{H}_V^U$ given by (f, g, α) where $f : X \to U$, $g : V \to Y$ and $\alpha_{x,v} : \hom_R\left(\mathcal{G}_{gv}^x, \mathcal{H}_v^{fx}\right)$. We need to find an element of

$$\sum_{\substack{f':X\to U \\ g':X\times V\to Y^*}} \prod_{\substack{x:X \\ v:V}} \hom_S\left(\prod_{y\in g'(x,v)} M(\mathcal{G}_y^x), M(\mathcal{H}_v^{f'x})\right)$$

We take $f' = f$ and $g'(x, v)$ to be the multiset containing only gv. Then

$$\prod_{y\in g'(x,v)} M(\mathcal{G}_y^x) = M(\mathcal{G}_{gv}^x)$$

and so $M(\alpha_{x,v})$ is a morphism of the correct type.

Suppose the linearisation functor is $L : S \to R$. The functor $\mathfrak{D}_{dn}(L) : \mathfrak{D}_i(S) \to \mathfrak{D}_l(R)$ acts on objects $\mathcal{G}_Y^X$ by

$$(\mathfrak{D}_{dn}(L)(\mathcal{G}))_{X\to Y^*}^X : \binom{x}{f} \mapsto \bigotimes_{y\in fx} L(\mathcal{G}_y^x)$$

Multiplicatives

$$1^{\{*\}}_{\{*\}} : \binom{*}{*} \mapsto 1$$

$$\bot^{\{*\}}_{\{*\}} : \binom{*}{*} \mapsto \bot$$

$$(\mathcal{G}^X_Y)^\bot = (\mathcal{G}^\bot)^Y_X : \binom{y}{x} \mapsto (\mathcal{G}^x_y)^\bot$$

$$\mathcal{G}^X_Y \otimes \mathcal{H}^U_V = (\mathcal{G} \otimes \mathcal{H})^{X \times U}_{(U \to Y) \times (X \to V)} : \binom{x,u}{f,g} \mapsto \mathcal{G}^x_{fu} \otimes \mathcal{H}^u_{gx}$$

$$\mathcal{G}^X_Y \parr \mathcal{H}^U_V = (\mathcal{G} \parr \mathcal{H})^{(V \to X) \times (Y \to U)}_{Y \times V} : \binom{f,g}{y,v} \mapsto \mathcal{G}^{fv}_y \parr \mathcal{H}^{gy}_v$$

Additives

$$\top^{\{*\}}_{\varnothing}$$

$$0^{\varnothing}_{\{*\}}$$

$$\mathcal{G}^X_Y \mathbin{\&} \mathcal{H}^U_V = (\mathcal{G} \mathbin{\&} \mathcal{H})^{X \times U}_{Y + V} : \binom{x,u}{z} \mapsto \begin{cases} \mathcal{G}^x_z & \text{if } z \in Y \\ \mathcal{H}^u_z & \text{if } z \in V \end{cases}$$

$$\mathcal{G}^X_Y \oplus \mathcal{H}^U_V = (\mathcal{G} \oplus \mathcal{H})^{X + U}_{Y \times V} : \binom{z}{y,v} \mapsto \begin{cases} \mathcal{G}^z_y & \text{if } z \in X \\ \mathcal{H}^z_v & \text{if } z \in U \end{cases}$$

Exponentials

$$!\mathcal{G}^X_Y = (!\mathcal{G})^X_{X \to Y^*} : \binom{x}{f} \mapsto \bigotimes_{y \in fx} !\mathcal{G}^x_y$$

$$?\mathcal{G}^X_Y = (?\mathcal{G})^{Y \to X^*}_Y : \binom{g}{y} \mapsto \parr_{x \in gy} ?\mathcal{G}^x_y$$

Figure 1 Interpretation of constants and connectives in $\mathfrak{D}_l(R)$.

Here $\bigotimes_{y \in fx}$ is the fold of the monoidal product of R over the finite multiset fx, where the fold over the empty multiset is the unit $1 \in R$. The subscript dn stands for *Diller-Nahm*, since this definition contains the essence of the Diller-Nahm functional interpretation. The intuitive justification for this definition is that the exponential $\mathfrak{D}_{dn}(L) \circ \mathfrak{D}_f(M)$ should be an interpretation of

$$!\forall y \in fx.\, |\varphi|^x_y$$

which is the Diller-Nahm interpretation of the exponentials in [24]. Since we are working over set theory we 'know' the (finite) size of fx, so we can replace the $\forall$ with a folded &. (This is a subtle point: we are simply defining a family of formulas, whereas when using free variables a formula must have a fixed structure.) Then we use the fact that ! is strong monoidal (the 'transmutation principle' of linear logic, see section 7.1 of [22]) to obtain

$$\bigotimes_{y \in fx} ! \, |\varphi|^x_y$$

When this is factored as

$$\bigotimes_{y \in fx} L \left(M \, |\varphi|^x_y \right)$$

the M becomes absorbed into the definition of $\mathfrak{D}_f(M)$, and we are left with $\mathfrak{D}_{dn}(L)$. (We could write it instead as $L\&$, but using $\otimes L$ gives the exponential in Figure 1 directly. Taking the exponential to be $\otimes !$ is preferable to $!\&$ because we need not assume that L has products.)

Now suppose we have a morphism of $\mathfrak{D}_i(S)$ from $\mathcal{G}^X_Y$ to $\mathcal{H}^U_V$ given by (f, g, α) where $f : X \to U$, $g : X \times V \to Y^*$ and

$$\alpha_{x,v} : \hom_S \left(\prod_{y \in g(x,v)} \mathcal{G}^x_y, \mathcal{H}^{fx}_v \right)$$

We need to find an element of

$$\hom_{\mathfrak{D}_l(R)} \left((\mathfrak{D}_{dn}(L)(\mathcal{G}))^X_{X \to Y^*}, (\mathfrak{D}_{dn}(L)(\mathcal{H}))^U_{U \to V^*} \right)$$

The witnesses are $f : X \to U$ and $g' : (U \to V^*) \to (X \to Y^*)$ given by

$$g' = \lambda h^{U \to V^*}, x^X . h(fx) \gg\!= \lambda v^V . g(x, v)$$

Given $x \in X$ and $h : U \to V^*$ we need to find an element of

$$\hom_R \left((\mathfrak{D}_{dn}(L)(\mathcal{G}))^x_{g'h}, (\mathfrak{D}_{dn}(L)(\mathcal{H}))^{fx}_h \right) = \hom_R \left(\bigotimes_{y \in g'hx} L(\mathcal{G}^x_y), \bigotimes_{v \in h(fx)} L(\mathcal{H}^{fx}_v) \right)$$

We have

$$\bigotimes_{v \in h(fx)} L(\alpha_{x,v}) : \hom_R \left(\bigotimes_{v \in h(fx)} L \left(\prod_{y \in g(x,v)} \mathcal{G}^x_y \right), \bigotimes_{v \in h(fx)} L(\mathcal{H}^{fx}_v) \right)$$

Here we can use that L is a symmetric monoidal functor to get an element of

$$\hom_R \left(\bigotimes_{v \in h(fx)} \bigotimes_{y \in g(x,v)} L(\mathcal{G}^x_y), \bigotimes_{v \in h(fx)} L(\mathcal{H}^{fx}_v) \right)$$

Finally the left hand side can be written as a single monoidal product over $y \in g'hx$ by definition of the monadic bind.

▶ **Lemma 4.** *$\mathfrak{D}_{dn}(L) \dashv \mathfrak{D}_f(M) : \mathfrak{D}_l(R) \to \mathfrak{D}_i(S)$ is a linear-nonlinear adjunction.*

Proof. By proposition 14 of [22] it suffices to prove that $\mathfrak{D}_{dn}(L) \dashv \mathfrak{D}_f(M)$ is an adjunction and $\mathfrak{D}_{dn}(L)$ is strong symmetric monoidal.

The equation for the adjunction is

$$\hom_{\mathfrak{D}_l(R)}\left(\mathfrak{D}_{dn}(L)(\mathcal{G}_Y^X), \mathcal{H}_V^U\right) \cong \hom_{\mathfrak{D}_i(S)}\left(\mathcal{G}_Y^X, \mathfrak{D}_f(M)(\mathcal{H}_V^U)\right)$$

We evaluate

$$\hom_{\mathfrak{D}_l(R)}\left(\mathfrak{D}_{dn}(L)(\mathcal{G}_Y^X), \mathcal{H}_V^U\right) = \sum_{\substack{f:X\to U \\ g:V\to(X\to Y^*)}} \prod_{\substack{x:X \\ v:V}} \hom_R\left(\bigotimes_{y\in gvx} L(\mathcal{G}_y^x), \mathcal{H}_v^{fx}\right)$$

and

$$\hom_{\mathfrak{D}_i(S)}\left(\mathcal{G}_Y^X, \mathfrak{D}_f(M)(\mathcal{H}_V^U)\right) = \sum_{\substack{f:X\to U \\ g:X\times V\to Y^*}} \prod_{\substack{x:X \\ v:V}} \hom_S\left(\prod_{y\in g(x,v)} \mathcal{G}_y^x, M(\mathcal{H}_v^{fx})\right)$$

These are isomorphic using $L \dashv M$ and the fact that L is strong monoidal.

To prove that $\mathfrak{D}_{dn}(L)$ is strong monoidal we must show that

$$\mathfrak{D}_{dn}(L)(\mathcal{G}_Y^X) \otimes \mathfrak{D}_{dn}(L)(\mathcal{H}_V^U) \cong \mathfrak{D}_{dn}(L)(\mathcal{G}_Y^X \mathbin{\&} \mathcal{H}_V^U)$$

We evaluate

$$(\mathfrak{D}_{dn}(L)(\mathcal{G}) \otimes \mathfrak{D}_{dn}(L)(\mathcal{H}))^{X\times U}_{(X\times U\to Y^*)\times(X\times U\to V^*)} : \begin{pmatrix} x,u \\ f,g \end{pmatrix} \mapsto \bigotimes_{y\in f(x,u)} L(\mathcal{G}_y^x) \otimes \bigotimes_{v\in g(x,u)} L(\mathcal{H}_v^u)$$

and

$$\mathfrak{D}_{dn}(L)(\mathcal{G} \mathbin{\&} \mathcal{H})^{X\times U}_{X\times U\to(Y+V)^*} : \begin{pmatrix} x,u \\ h \end{pmatrix} \mapsto \bigotimes_{z\in h(x,u)} \begin{cases} L(\mathcal{G}_z^x) & \text{if } z \in Y \\ L(\mathcal{H}_z^u) & \text{if } z \in V \end{cases}$$

These are isomorphic due to the natural isomorphism $Y^* \times V^* \cong (Y+V)^*$ (note that this isomorphism does not hold if we replace finite multisets with finite ordered lists, ie. free commutative monoids by free noncommutative monoids). Finally, the symmetry of $\mathfrak{D}_{dn}(L)$ also inherits easily from that of L. ◀

We can therefore derive the interpretation of ! as the composition $\mathfrak{D}_{dn}(L) \circ \mathfrak{D}_f(M)$. Given an object $\mathcal{G}_y^x$, its exponential is

$$(!\mathcal{G})^X_{X\to Y^*} : \begin{pmatrix} x \\ f \end{pmatrix} \mapsto \bigotimes_{y\in fx} !\mathcal{G}_y^x$$

where the exponential in the underlying model is $! = L \circ M$.

It is worth noting that, as in chapter 4 of [9], the functor $\mathfrak{D}_{dn}(L)$ factors into three parts $\mathfrak{D}_{dn}(L) = B \circ A \circ \mathfrak{D}_f(L)$ where A and B (called T and S in [9]) are endofunctors on $\mathfrak{D}_l(R)$ given respectively by

$$(A(\mathcal{G}))^X_{Y^*} : \begin{pmatrix} x \\ s \end{pmatrix} \mapsto \bigotimes_{y\in s} \mathcal{G}_y^x$$

and

$$(B(\mathcal{G}))^X_{X\to Y} : \binom{x}{f} \mapsto \mathcal{G}^x_{fx}$$

We can interpret A and B game-semantically as giving two different advantages to Abelard. A allows Abelard to play several moves, and B allows Abelard to observe Eloise's move. Both of these are expressed by monads on the category of sets, respectively the finite multiset monad and the reader monad $(X \to)$. The exponential of $\mathfrak{D}_l(R)$ therefore factors into four parts as

$$B \circ A \circ \mathfrak{D}_f(L) \circ \mathfrak{D}_f(M)$$

The functors A and B have much structure in their own right: they are both comonads on $\mathfrak{D}_l(R)$ with a distributivity law between them making $B \circ A$ into another comonad. However $B \circ A$ is a linear exponential comonad (which is a direct categorical semantics of the exponential, see [16]), whereas A and B individually are not. The entire reason we also compose with $\mathfrak{D}_f(L) \circ \mathfrak{D}_f(M) = \mathfrak{D}_f(L \circ M)$, which after all requires more structure in the underlying model, is to obtain the completeness theorem in section 5.

The lemmas in this section add up to a soundness theorem.

▶ **Theorem 5.** *If $\mathcal{R}$ is a sound model of* **MELL** *then $\mathfrak{D}(\mathcal{R})$ is a sound model of* **LL**.

4 Games with bidding

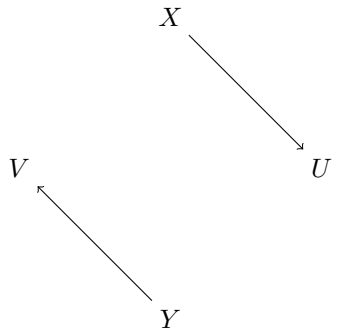
In section 5 we will investigate applying the transformation $\mathfrak{D}$ to models which are complete (that is truth implies provability, which is a weaker property than *full completeness* which is more often considered). In practice this means letting $\mathcal{R}$ be a game model. In this section we give some general remarks about $\mathfrak{D}(\mathcal{R})$ when $\mathcal{R}$ is a game model.

In general, a game model is a category R whose objects are games, and whose morphisms are (relative) winning strategies. Thus logically formulas are denoted by games and proofs by winning strategies. The denotation of linear negation is interchange of players (at least for classical linear logic), and the denotation of $\otimes$ is some form of concurrent play, making R into a *-autonomous category. For models which have additives the product $\mathcal{G} \,\&\, \mathcal{H}$ is usually denoted by a game in which Abelard chooses which of the two games will be played, and for $\mathcal{G} \oplus \mathcal{H}$ Eloise makes the choice. The exponential is often similar to an infinite tensor product. The point of making these informal observations is that they are preserved under the transformation $\mathfrak{D}$.

We begin by considering the two-element boolean algebra $\mathbb{B}$ as a degenerate game model containing only two games: one which Eloise wins immediately, and one which Abelard wins immediately. Thus we can see $\mathfrak{D}(\mathbb{B})$, which is called $\mathbf{G}(\mathcal{C})$ in the terminology of [9] (where $\mathcal{C}$ is the category of sets or another suitable model of dependent type theory), as a model of games with bidding in which the games contain only the bidding round, and after the bidding round one player is declared to have won. The possibility of viewing dialectica categories as categories of games has been discussed in several places, and in particular in the final section of [6], and this section greatly extends that idea.

One issue with viewing dialectica categories as games is the strange 'causality' in a game such as $\mathcal{G}^X_Y \multimap \mathcal{H}^U_V$, in which u depends on x but not v, and y depends on v but not x. One way to view the strange dynamics of this game is as a generalisation of history-freeness in which the moves are chosen in the order (x, u, v, y), where Abelard's strategy to choose x, v is history-free and Eloise's strategy may depend on the most recent move but not the remainder

of the history. Alternatively we can imagine the bidding round to be played by two teams of two players (like in bridge) with a particular message-passing protocol



Partners sit opposite each other, with X and Y representing Abelard and U and V representing Eloise, and the arrows representing the direction of message-passing. Unfortunately both of these intuitions (history-freeness and message-passing) break down when we consider higher order bids (that is, bids which are functions depending on other functions). There is a general but less satisfactory intuition in these cases: the players submit (higher order) computer programs, which are finite representations of their strategy, to play on their behalf.

Now we consider informally a 'general non-degenerate game model'. The conclusion is that the construction $\mathfrak{D}$, which can be applied to any model, preserves the property of 'being a game model'. For a concrete game model these informal remarks could be made precise: the simplest example is the category of Blass games of [6]; in section 5 we consider the category of Hyland-Ong games [15] and the category of asynchronous games of [20].

An object $\mathcal{G}_Y^X$ of $\mathfrak{D}_l(R)$ consists of sets of bids X and Y for Eloise and Abelard, together with a game $\mathcal{G}_y^x$ in the underlying model for each pair of bids. Thus a winning strategy for Eloise consists of a bid $x \in X$, together with a winning strategy σ_y for $\mathcal{G}_y^x$ for every bid y of Abelard. Thus $\mathcal{G}_Y^X$ can be seen as a game with bidding: first Eloise and Abelard simultaneously bid, and then the pair of chosen bids determines precisely which subsequent game will be played. (Very informally this is somewhat like the game of bridge: there is an initial bidding round which determines exactly which variant of whist will be played.)

The negation of $\mathfrak{D}_l(R)$ is to interchange players in the bidding round and then apply the negation of R. Thus when R is a game model the negation of $\mathfrak{D}_l(R)$ overall is simply interchange of players in the compound game. The other connectives which behave very cleanly are the additives: they are similar to the additives in a general game model except that the choice of which game to play occurs *simultaneously* with the other bids. Thus for example in the game $\mathcal{G}_Y^X \mathbin{\&} \mathcal{H}_V^U$ Abelard chooses a game and a bid for that game, but since Eloise bids simultaneously she must choose a bid for both games. Thus a winning strategy for Eloise in $\mathcal{G}_Y^X \mathbin{\&} \mathcal{H}_V^U$ consists of a pair of bids (x, u) together with winning strategies for both $\mathcal{G}_y^x$ and $\mathcal{H}_v^u$.

The denotation of the tensor product $\mathcal{G}_Y^X \otimes \mathcal{H}_V^U$ is more complicated. Eloise simply bids a pair (x, u). Simultaneously Abelard must bid a pair of functions $f : U \to Y$ and $g : X \to V$, and then the games $\mathcal{G}_{fu}^x$ and $\mathcal{H}_{gx}^u$ are played in parallel in the sense specified by R. Similarly the exponential $!\mathcal{G}_Y^X$ is played as follows. Firstly Eloise chooses a bid $x \in X$. Then Abelard observes this and chooses a finite multiset $y_1, \ldots, y_n \in Y$. For each y_i there is an exponential $!\mathcal{G}_{y_i}^x$, which will be similar to the parallel composition of infinitely many copies of $\mathcal{G}_{y_i}^x$. Then each of the $!\mathcal{G}_{y_i}^x$ is played in parallel, but typically a different sense of parallel than is used for exponentials. Since the notion of winning strategy for Eloise in these games will depend on exactly what notion of parallelism is used in R, it is difficult to say more in general.

As explained above, in some cases it is possible to consider this as a game played by two pairs of partners with a message-passing protocol, but in general it is necessary to consider functions which can depend on other functions in a higher-order way. Thus from a game-semantic perspective it will be more satisfying to replace the category of sets with a different locally cartesian closed category in which functions contain only a finite amount of information, such as a coherence space model of type theory [23]. Particularly interesting would be to link to recent work in progress of Abramsky, Jagadeesan and others on game semantics of dependent type theory. (These models are closely related to recent work on linear dependent types [29].) This would lead to a two-layered game model in which the bidding round has finer structure and the bids themselves specify strategies for sub-games. The difficulty would be to find a suitable sense in which R is enriched and fibered over the model of dependent types.

5 Relative completeness for additive-free fragments

▶ **Definition 6** (Complete model)**.** Let $\mathcal{R}$ be a model of **LL**. A mapping from atoms to objects of R is called a valuation in $\mathcal{R}$. Given a valuation v, we can extend it inductively to an interpretation of formulas in $\mathcal{R}$, denoted $[\![\varphi]\!]_v$ or simply $[\![\varphi]\!]$.

$\mathcal{R}$ is called a complete model of **LL** if for all formulas φ, ψ, if $\hom_R([\![\varphi]\!]_v, [\![\psi]\!]_v)$ is nonempty for all valuations v then the sequent $\varphi \vdash \psi$ is derivable in **LL**. Completeness for **MELL** and other fragments is defined similarly.

A *characterisation theorem* for a functional interpretation is a result saying that the equivalence between φ and its functional interpretation $\exists x \forall y. |\varphi|^x_y$ is derivable in some system, usually a base language like $\mathbf{HA}^\omega$ extended with *characterisation principles*, which are axioms validated by the functional interpretation such as the axiom of choice, Markov's principle and independence of premise. In order to obtain the statement of the following lemma we take the logical formula

$$\varphi \leftrightarrow \exists x \forall y. |\varphi|^x_y$$

and split the bi-implication into its defining conjunction, then in each part we prenex the quantifiers and interpret them as dependent types. (Special care would need to be paid to these manipulations if the category of sets was replaced by a different model, as suggested in the previous section.)

The characterisation theorem for classical linear logic in [25] uses not $\exists x \forall y$ but a Henkin quantifier, $\exists\!\!\forall^x_y$, and so this 'rearrangement' is unsound. The result we see is that this lemma fails to extend from **MELL** to **LL**. (Given that this simultaneity is at the heart of the functional interpretations of classical linear logic, it is remarkable that this method works at all.) See section 7 for a discussion of how to extend the completeness theorem to include additives by correctly interpreting the simultaneous quantifier.

▶ **Lemma 7.** *Let $\mathcal{R}$ be a model of* **MELL** *and let v be a valuation in $\mathcal{R}$. Let φ be a formula of* **MELL** *with interpretation $|\varphi|^X_Y$ in $\mathfrak{D}(\mathcal{R})$, where the interpretation of an atomic proposition is*

$$|p|^{\{*\}}_{\{*\}} : \begin{pmatrix} * \\ * \end{pmatrix} \mapsto v(p)$$

Then the types

$$\sum_{x:X} \prod_{y:Y} \hom_R([\![\varphi]\!], |\varphi|^x_y)$$

and

$$\sum_{y:Y} \prod_{x:X} \hom_R(|\varphi|_y^x, [\![\varphi]\!])$$

are inhabited.

Proof. These are proved simultaneously by induction on φ. In the base case we have $\varphi = p$ is an atom, and the point $*$ and identity morphism witnesses both (1) and (2).

In the negation case for (1) the inductive hypothesis for (2) gives $y \in Y$ together with morphisms $\pi_x : \hom_R(|\varphi|_y^x, [\![\varphi]\!])$. Then $\pi_x^\perp : \hom_R([\![\varphi^\perp]\!], |\varphi^\perp|_x^y)$. The case for (2) is symmetric.

For (1) of $\otimes$ the inductive hypothesis gives x and u together with morphisms $\pi_y : \hom_R([\![\varphi]\!], |\varphi|_y^x)$ and $\sigma_v : \hom_R([\![\psi]\!], |\psi|_v^u)$. Then for each $f : U \to Y$ and $g : X \to V$ we have

$$\pi_{fu} \otimes \sigma_{gx} : \hom_R\left([\![\varphi \otimes \psi]\!], |\varphi \otimes \psi|_{f,g}^{x,u}\right)$$

For (2) of $\otimes$ the inductive hypothesis gives y and v together with morphisms $\pi_x : \hom_R(|\varphi|_y^x, [\![\varphi]\!])$ and $\sigma_u : \hom_R(|\psi|_v^u, [\![\psi]\!])$. Define $f : U \to Y$ by $fu = y$ and $g : X \to V$ by $gx = v$. Then for each (x, u) we have

$$\pi_x \otimes \sigma_u : \hom_R\left(|\varphi \otimes \psi|_{f,g}^{x,u}, [\![\varphi \otimes \psi]\!]\right)$$

For (1) of !, by the inductive hypothesis we have x together with morphisms in $\pi_y : \hom_R([\![\varphi]\!], |\varphi|_y^x)$. Let $f : X \to Y^*$. We have

$$\bigotimes_{y \in fx} !\pi_y : \hom_R\left(\bigotimes_{y \in fx} ![\![\varphi]\!], \bigotimes_{y \in fx} ! |\varphi|_y^x\right)$$

Since $\mathcal{R}$ is a model of **MELL** we have an inhabitant of

$$\hom_R\left(![\![\varphi]\!], \bigotimes_{y \in fx} ![\![\varphi]\!]\right)$$

and we are done.

For (2) of !, by the inductive hypothesis we have y together with morphisms $\pi_x : \hom_R(|\varphi|_y^x, [\![\varphi]\!])$. Take f to be the constant function returning the singleton multiset containing y. Then we have

$$!\pi_x : \hom_R(\bigotimes_{y \in fx} ! |\varphi|_y^x, ![\![\varphi]\!])$$

and we are done. ◀

(This lemma would be much less interesting if we used the dialectica rather than the Diller-Nahm exponential, because in that case X and Y would always have size 0 or 1. The Diller-Nahm interpretation of **MELL**, on the other hand, allows interesting sets such as $\mathbb{N} = \{*\}^*$ and $\mathbb{R} = \mathbb{N} \to \mathbb{N}$.)

▶ **Theorem 8** (Relative completeness). *Let $\mathcal{R}$ be a model of* **MELL** *and let φ be a formula of* **MELL** *which is true in $\mathfrak{D}(\mathcal{R})$. Then φ is true in $\mathcal{R}$.*

Proof. Let v be a valuation in $\mathcal{R}$, and let φ be a formula of **MELL** with interpretation $|\varphi|^X_Y$ in $\mathfrak{D}(\mathcal{R})$ using the same interpretation of atomic propositions defined in the lemma. Since φ is true in $\mathfrak{D}(\mathcal{R})$ we have a winning bid $x : X$ together with winning strategies

$$\pi_y : \hom_R(1, |\varphi|^x_y)$$

From (2) of the lemma we have $y : Y$ together with winning strategies

$$\sigma_x : \hom_R(|\varphi|^x_y, [\![\varphi]\!])$$

Therefore

$$\sigma_x \circ \pi_y : \hom_R(1, [\![\varphi]\!])$$

Since this holds for every valuation, φ is true in $\mathcal{R}$. ◀

Let **HO** be the category of Hyland-Ong games and history-free, uniformly winning strategies [15], with the the identity functor considered as an exponential. Then $\mathfrak{D}(\mathbf{HO})$ is the model of 'Hyland-Ong games with bidding'. (As a linear-nonlinear adjunction, the model of Hyland-Ong games has $R = S = \mathbf{HO}$, and $L = M$ is the identity functor.)

▶ **Corollary 9.** $\mathfrak{D}(\mathbf{HO})$ *is a sound model of* **LL** *and a complete model of* **MLL**.

Notice that because the posetal reflection of **HO** is a *lineale* in the sense of [8] (including having a trivial exponential), the category $\mathfrak{D}(\mathbf{HO})$ is an example of the construction in that paper (modulo size issues). However examples of this kind have not been considered before, and in particular the completeness result is new.

Let **AG** be the category Z of asynchronous games and (equivalence classes of) innocent winning strategies [20]. This is a sound model of **LL** which is proven in [21] to be complete for **MELL**. That paper also provides a small variation which is complete for **LL**, although using that model will not be necessary for our purposes.

▶ **Corollary 10.** $\mathfrak{D}(\mathbf{AG})$*, the category of asynchronous games with bidding, is a sound model of* **LL** *and a complete model of* **MELL**.

A large part of the motivation for this paper is to introduce the category $\mathfrak{D}(\mathbf{AG})$ and prove its soundness. It is an interesting model which will be studied in detail by the author in a follow-up paper: in particular there is a way to analyse the formulas containing additives which are valid in the model. See section 7 for a summary of the argument.

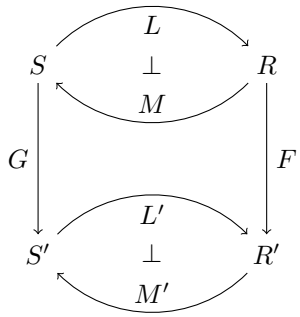
6 $\mathfrak{D}$ is a functor

Given a model $\mathcal{R}$ of **MELL**, presented as a linear-nonlinear adjunction, we have defined a model $\mathfrak{D}(\mathcal{R})$ of **LL**. Since a collection of models forms a category we can ask whether $\mathfrak{D}$ is a functor. The answer is 'yes' for the strongest notion of a morphism of models: a pair of functors which commute with all of our structure. Results of this kind are standard, and appear as early as [27].

Since models are pairs of structured categories, they moreover form a 2-category, with 1-cells given by pairs of monoidal functors satisfying suitable conditions, and 2-cells given by pairs of natural transformations. We will leave the consideration of 2-categorical issues for later work, but it should be noted that most of the diagrams in this section commute only up to natural isomorphism.

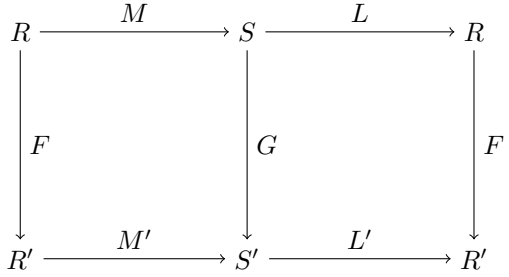
This section does not contain all cases of the proofs (which would take another paper), but highlight the most interesting cases. Most of the proofs amount to showing that certain (sometimes quite formidable) dependent types are inhabited, and thus are natural candidates for formalisation in a dependently typed programming language, with suitable libraries for monoidal category theory and 2-category theory. The author intends to carry this out in the future.

▶ **Definition 11** (Morphism of linear-nonlinear adjunctions)**.** Let $L \dashv M : R \to S$ and $L' \dashv M' : R' \to S'$ be linear-nonlinear adjunctions. A morphism (F, G) from the former to the latter consists of functors

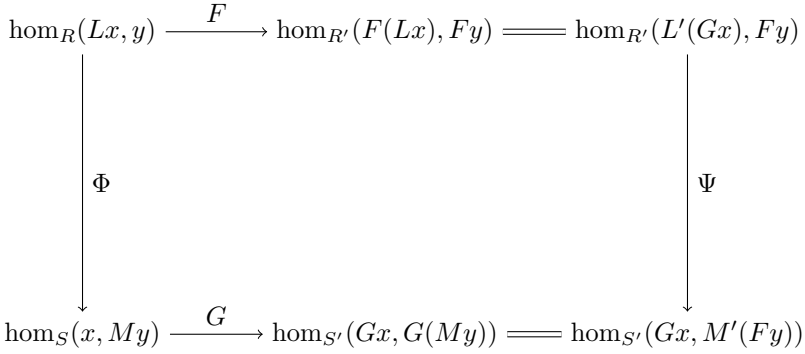


such that

1. F is a monoidal functor
2. F and G are cartesian monoidal functors
3. The following diagram commutes:



4. The following diagram commutes:



(where Φ, Ψ are the isomorphisms associated to the adjunctions $L \dashv M$ and $L' \dashv M'$).

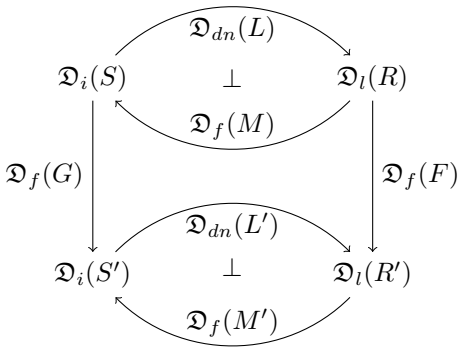
The category of linear-nonlinear adjunctions and morphisms will be called **LL-Mod**. The (larger) category of linear-nonlinear adjunctions in which R and R' do not necessarily

have products (and F is not necessarily cartesian monoidal) will be called **MELL-Mod**. There is a forgetful functor $U : \mathbf{LL}\text{-}\mathbf{Mod} \to \mathbf{MELL}\text{-}\mathbf{Mod}$.

This definition is based on the 'maps of adjunctions' in [18]. The equivalent definitions for the intuitionistic variants are given in [19]. If we weaken this to having natural transformations $M' \circ F \implies G \circ M$ and $L' \circ G \implies F \circ L$ we obtain the linear-nonlinear equivalent of the 'map of models' of [16].

▶ **Lemma 12.** *$\mathfrak{D}$ is a functor* $\mathbf{MELL}\text{-}\mathbf{Mod} \to \mathbf{LL}\text{-}\mathbf{Mod}$.

Proof. We need to prove that



is a morphism of **LL-Mod**, given that (F, G) is a morphism of **MELL-Mod**.

We will prove the conditions for exponentials, namely that we have commuting squares

$$\begin{array}{ccccc}
\mathfrak{D}_l(R) & \xrightarrow{\mathfrak{D}_f(M)} & \mathfrak{D}_i(S) & \xrightarrow{\mathfrak{D}_{dn}(L)} & \mathfrak{D}_l(R) \\
\downarrow{\scriptstyle \mathfrak{D}_f(F)} & & \downarrow{\scriptstyle \mathfrak{D}_f(G)} & & \downarrow{\scriptstyle \mathfrak{D}_f(F)} \\
\mathfrak{D}_l(R') & \xrightarrow{\mathfrak{D}_f(M')} & \mathfrak{D}_i(S') & \xrightarrow{\mathfrak{D}_{dn}(L')} & \mathfrak{D}_l(R')
\end{array}$$

For the left hand square let $\mathcal{G}^X_Y \in \mathfrak{D}_l(R)$. We have

$$((\mathfrak{D}_f(M') \circ \mathfrak{D}_f(F))(\mathcal{G}))^X_Y : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto (M' \circ F)(\mathcal{G}^x_y)$$

$$((\mathfrak{D}_f(G) \circ \mathfrak{D}_f(M))(\mathcal{G}))^X_Y : \begin{pmatrix} x \\ y \end{pmatrix} \mapsto (G \circ M)(\mathcal{G}^x_y)$$

These are equivalent using the identity functions on X and Y and the natural isomorphism $M' \circ F \cong G \circ M$. For the right hand square let $\mathcal{G}^X_Y \in \mathfrak{D}_i(S)$. Then we have

$$((\mathfrak{D}_{dn}(L') \circ \mathfrak{D}_f(G))(\mathcal{G}))^X_{X \to Y^*} : \begin{pmatrix} x \\ f \end{pmatrix} \mapsto \bigotimes_{y \in fx} (L' \circ G)(\mathcal{G}^x_y)$$

$$((\mathfrak{D}_f(F) \circ \mathfrak{D}_{dn}(L))(\mathcal{G}))^X_{X \to Y^*} : \begin{pmatrix} x \\ f \end{pmatrix} \mapsto F\left(\bigotimes_{y \in fx} L(\mathcal{G}^x_y)\right)$$

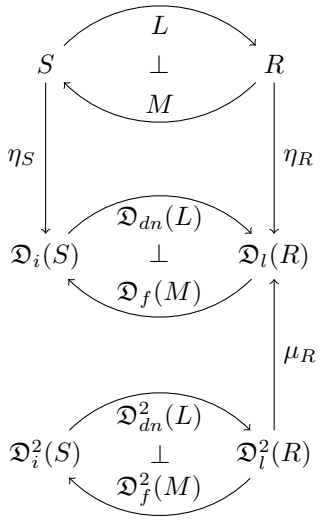


Figure 2 Unit and multiplication of $\mathfrak{D}$.

Using the identity functions on X and $X \to Y^*$ together with the natural isomorphism $L' \circ G \cong F \circ L$ and the fact that F is monoidal we have natural transformations

$$\bigotimes_{y \in fx} (L' \circ G)(\mathcal{G}^x_y) \cong \bigotimes_{y \in fx} (F \circ L)(\mathcal{G}^x_y) \cong F\left(\bigotimes_{y \in fx} L(\mathcal{G}^x_y)\right)$$

◀

We have defined $\mathfrak{D}$ as a functor **MELL-Mod** $\to$ **LL-Mod**. By composing with the forgetful functor in the opposite direction we obtain an endofunctor on **MELL-Mod**. Next we will investigate a monad-like structure on $\mathfrak{D}$. The starting point is the observation that there is a family of functors $\mu_R : \mathfrak{D}^2_l(R) \to \mathfrak{D}_l(R)$, motivated by logical considerations in the next section, which appears to be the multiplication of a monad. We investigate this structure and show that, on the contrary, $\mathfrak{D}$ is not a monad. The functors μ_R behave badly with respect to exponentials, and the corresponding functors $\mu_S : \mathfrak{D}^2_i(S) \to \mathfrak{D}_i(S)$ cannot be defined in a reasonable way. The resulting setup is illustrated in Figure 2. Even when restricting to just **MLL**, the functors μ_R are only lax monoidal, and the second monad law fails to hold, even in a lax way.

The main theorem of [12], which gives a sense in which the dialectica interpretation is a pseudo-monad, is extremely closely related. There are two main differences, other than the fact that our dialectica categories are far less general. The first is that Hofstra's multiplication operator, from a game-semantic point of view, treats the two players asymmetrically, and so appears to be incompatible with classical linear logic. The second is that, by using linear-nonlinear semantics, we insist on soundness for linear logic with exponentials. Nevertheless the second monad law does not appear to rely on either of these facts, which implies that the constructions are more different than they appear.

▶ **Lemma 13.** *The functor*

$$\eta_R : R \to \mathfrak{D}_l(R)$$

which takes an object $x \in R$ to the game with one play and outcome x,

$$(\eta_R(x))^{\{*\}}_{\{*\}} : \binom{*}{*} \mapsto x$$

extends to a natural transformation $\mathbf{I} \to \mathfrak{D}$.

Next we explicitly find $\mathfrak{D}^2(\mathcal{R})$ as a model of **MELL**. An object $\mathcal{G}^X_Y$ of $\mathfrak{D}^2_l(R)$ consists of sets X and Y together with a family of objects $\mathcal{G}^x_y$ of $\mathfrak{D}_l(R)$. Each such $\mathcal{G}^x_y$ itself has the form $(\mathcal{G}^x_y)^{U^x_y}_{V^x_y}$, where U^x_y and V^x_y are families of sets dependent on x and y, and we have a family of objects $(\mathcal{G}^x_y)^u_v$ of R. This defines the objects of both categories $\mathfrak{D}^2_l(R)$ and $\mathfrak{D}^2_i(S)$.

Consider objects $\mathcal{G}^X_Y$ and $\mathcal{H}^W_Z$ of $\mathfrak{D}^2_l(R)$ given by $(\mathcal{G}^x_y)^{U^x_y}_{V^x_y}$ and $(\mathcal{H}^w_z)^{P^w_z}_{Q^w_z}$, and consider a morphism from $\mathcal{G}$ to $\mathcal{H}$. This consists of functions $f : X \to W$ and $g : Z \to Y$ together with morphisms from $\mathcal{G}^x_{gz}$ to $\mathcal{H}^{fx}_z$ in $\mathfrak{D}_l(R)$. Each such morphism itself consists of functions $\alpha : U^x_{gz} \to P^{fx}_z$ and $\beta : Q^{fx}_z \to V^x_{gz}$ together with morphisms in R. Thus we have

$$\hom_{\mathfrak{D}^2_l(R)}(\mathcal{G}, \mathcal{H}) = \sum_{\substack{f:X\to W\\ g:Z\to Y}} \prod_{x:X} \sum_{\substack{\alpha:U^x_{gz}\to P^{fx}_z\\ \beta:Q^{fx}_z\to V^x_{gz}}} \prod_{u:U^x_{gz}} \hom_R\left((\mathcal{G}^x_{gz})^u_{\beta q}, (\mathcal{H}^{fx}_z)^{\alpha u}_q\right)$$

By thinking of $\mathfrak{D}^2_l(R)$ as a game model the definition of μ_R becomes obvious. We begin with a game model R of **MLL**, and prepend a bidding round to obtain $\mathfrak{D}_l(R)$, then prepend an earlier bidding round to obtain $\mathfrak{D}^2_l(R)$. A strategy for a game in this model consists of a bid in the first bidding round, together with a bid in the second bidding round for each possible bid of the opponent, and finally a strategy for each resulting game. This can be converted into a game with a single bidding round by bidding dependent types. Formally, given $\mathcal{G}^X_Y$ in $\mathfrak{D}^2_l(R)$ given by $(\mathcal{G}^x_y)^{U^x_y}_{V^x_y}$, we define the object $\mu_R(\mathcal{G})$ of $\mathfrak{D}_l(R)$ by

$$(\mu_R(\mathcal{G}))^{\sum_{x:X}\prod_{y:Y} U^x_y}_{\sum_{y:Y}\prod_{x:X} V^x_y} : \binom{x, f}{y, g} \mapsto (\mathcal{G}^x_y)^{fy}_{gx}$$

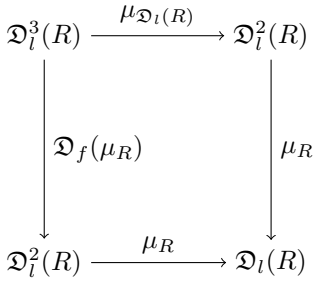
The functors μ_R are lax monoidal but not strong monoidal, and they do not commute with exponentials, even in a lax way. The linear-nonlinear semantics gives us a better perspective on this problem. We can think of objects of $\mathfrak{D}_i(S)$ as games with bidding, but in which in the bidding round Abelard has the advantages granted by the exponential, namely he can observe Eloise's move and then choose several possible moves. In particular, the sequentiality of the bidding prevents us from extending our intuition about μ_R to $\mathfrak{D}^2_i(S)$. A compound game in $\mathfrak{D}^2_i(S)$ has two bidding rounds which are each played sequentially, and so bids are made in the order $\exists\forall\exists\forall$. We cannot reduce this to a single round of dependent bidding, because there is no way to specify that Abelard's first bid cannot depend on Eloise's second bid.

Restricting to **MLL**, the first monad law holds up to natural isomorphism.

▶ **Theorem 14.** *There are natural isomorphisms*

$$\begin{array}{ccc}
\mathfrak{D}_l(R) & \xrightarrow{\eta_{\mathfrak{D}_l(R)}} & \mathfrak{D}^2_l(R) \\
\Big\downarrow{\scriptstyle \mathfrak{D}_f(\eta_R)} & \diagdown\!\!\diagdown & \Big\downarrow{\scriptstyle \mu_R} \\
\mathfrak{D}^2_l(R) & \xrightarrow{\mu_R} & \mathfrak{D}_l(R)
\end{array}$$

The second monad law



fails, even in a lax way (that is, this diagram does not contain a 2-cell).

7 Towards the additives

In this section we briefly look at the question of how the completeness result in section 5 should be extended to full **LL**. The intuition is that we are trying to simulate the behaviour of the simultaneous quantifier in [25], in order to find a better analogue to the characterisation theorem $\varphi \multimap \forall\!\!\!\forall_y^x |\varphi|_y^x$. This is ongoing work by the author, and this section only outlines the method.

We extend the language of **MELL** as follows. For a double-indexed family of formulas $|\varphi|_Y^X$ we freely add a formula called $\binom{\oplus x:X}{\&y:Y} |\varphi|_y^x$. These new formulas are called simultaneous additives (they could also be called 'Henkin additives', because simultaneous quantifiers are a special case of Henkin quantifiers). The definition is fully recursive, so the individual formulas $|\varphi|_y^x$ may themselves be simultaneous additives.

The intuition for simultaneous additives is exactly that for Henkin quantifiers in dialogue semantics. In dialogue semantics, for a folded additive disjunction $\oplus_{x:X}$ Eloise (the verifier) chooses x, and for a folded additive conjunction $\&_{y:Y}$ Abelard (the falsifier) chooses y. For the simultaneous additive $\binom{\oplus x:X}{\&y:Y}$ these choices are made simultaneously by the players.

There is a single introduction rule for simultaneous additives that captures this intuition. Suppose we have double-indexed families of formulas $|\varphi_i|_{Y_u}^{X_i}$ for $1 \leq i \leq m$ and $|\psi_j|_{V_j}^{U_j}$ for $1 \leq j \leq n$. For all functions

$$f_j : \prod_{i'} X_{i'} \times \prod_{j' \neq j} V_{j'} \to U_j$$
$$g_i : \prod_{i' \neq i} X_{i'} \times \prod_{j'} V_{j'} \to Y_i$$

for $1 \leq i \leq m$ and $1 \leq j \leq n$ we have a proof rule

$$\frac{\Gamma, \left(|\varphi_i|_{g_i(\vec{x}_{-i},\vec{v})}^{x_i}\right)_{i=1}^m \vdash \Delta, \left(|\psi_j|_{v_j}^{f_j(\vec{x},\vec{v}_{-j})}\right)_{j=1}^n \text{ for all } \vec{x} \in \prod_i X_i, \vec{v} \in \prod_j V_j}{\Gamma, \left(\binom{\oplus x_i:X_i}{\&y_i:Y_i} |\varphi_i|_{y_i}^{x_i}\right)_{i=1}^m \vdash \Delta, \left(\binom{\oplus u_j:U_j}{\&v_j:V_j} |\psi_j|_{v_j}^{u_j}\right)_{j=1}^n}$$

There is a hypothesis for all tuples $\vec{x}, \vec{v}$, hence this rule is generally infinitary. (The proof rule in [25] on which this is based uses free variables for $\vec{x}$ and $\vec{v}$ instead; it might be necessary to impose a restriction that the subproofs are 'uniform' in the parameters in some way.) The extended language will be called $\mathfrak{D}$ **LL**.

We extend the valuation of formulas in a model $\mathfrak{D}(\mathcal{R})$ to include simultaneous additives. If each $|\varphi|^x_y$ is a formula in the language of $\mathfrak{D}\mathbf{LL}$ with interpretation $\left||\varphi|^x_y\right|^{U^x_y}_{V^x_y}$ then the interpretation of $\binom{\oplus x:X}{\&y:Y}|\varphi|^x_y$ is given precisely by the μ operator:

$$\left|\binom{\oplus x:X}{\&y:Y}|\varphi|^x_y\right|^{\sum_{x:X}\prod_{y:Y}U^x_y}_{\sum_{y:Y}\prod_{x:X}V^x_y} : \binom{x,f}{y,g} \mapsto \left||\varphi|^x_y\right|^{fy}_{gx}$$

It is an open question what should be the semantics of simultaneous additives in an arbitrary category. If it exists, it must have properties of both a limit and a colimit, since it includes products and coproducts as special cases.

▶ **Theorem 15.** *Let R be any category, then $\mathfrak{D}_l(R)$ validates the simultaneous additive introduction rule.*

If we try to prove the equivalence of φ and $\binom{\oplus x:X}{\&y:Y}|\varphi|^x_y$, where φ is a formula of **LL**, we find that we need some additional principles beyond $\mathfrak{D}\,\mathbf{LL}$, corresponding to the characterising principles of a functional interpretation. Two of these are

$$\binom{\oplus x:X, u:U}{\&f:Y^U, g:V^X}|\varphi\otimes\psi|^{x,u}_{f,g} \multimap \binom{\oplus x:X}{\&y:Y}|\varphi|^x_y \otimes \binom{\oplus u:U}{\&v:V}|\psi|^u_v$$

and

$$\binom{\oplus z:X+U}{\&y:Y, v:V}|\varphi\oplus\psi|^z_{y,v} \multimap \binom{\oplus x:X}{\&y:Y}|\varphi|^x_y \oplus \binom{\oplus u:U}{\&v:V}|\psi|^u_v$$

The first is a propositional analogue of the *parallel choice* principle in [26], which itself is a generalisation of the independence of premise principle. Write $\mathfrak{D}\,\mathbf{LL}^{\#}$ for $\mathfrak{D}\,\mathbf{LL}$ extended with these axioms and others for the exponential. Then, by directly simulating the characterisation theorem for a functional interpretation it should be possible to prove that if $\mathcal{R}$ is sound and complete for **MELL** then $\mathfrak{D}(\mathcal{R})$ is sound and complete for $\mathfrak{D}\,\mathbf{LL}^{\#}$. In particular, $\mathfrak{D}(\mathbf{AG})$ should be a sound and complete model of $\mathfrak{D}\,\mathbf{LL}^{\#}$.

We continue by a purely syntactic argument. We prove that $\mathfrak{D}\,\mathbf{LL}$ has full cut elimination, and is a conservative extension of **LL** by identifying the usual additives with suitable simultaneous additives. Now if we take a formula φ in the language of **LL** which is validated by $\mathfrak{D}(\mathbf{AG})$, we know that φ is derivable in $\mathfrak{D}\,\mathbf{LL}^{\#}$, with a proof potentially involving both cuts and the characterising principles. In particular, since φ does not contain simultaneous additives, any simultaneous additives introduced in the proof by a characterising principle must be removed by a cut. By analysing the ways in which cut elimination can fail in the presence of characterising principles, it should be possible to identity axioms in the language of **LL** which are sound and complete for $\mathfrak{D}(\mathbf{AG})$. A simple example is the formula $\bot\otimes\top$, which is valid in every dialectica category but is not provable in **MELL**.

References

1 Jeremy Avigad and Solomon Feferman. Gödel's functional ("Dialectica") interpretation. In S. Buss, editor, *Handbook of proof theory*, volume 137 of *Studies in logic and the foundations of mathematics*, pages 337–405. North Holland, Amsterdam, 1998.

2 Michael Barr. *-autonomous categories and linear logic. *Mathematical structures in computer science*, 1(2):159–178, 1991.

3 Andrej Bauer. The Dialectica interpretation in Coq. Available electronically at `http://math.andrej.com/2011/01/03/the-dialectica-interpertation-in-coq/`, 2011.

4 P. N. Benton. A mixed linear and non-linear logic: proofs, terms and models (preliminary report). Technical report, University of Cambridge, 1994.

5 Bodil Biering. Cartesian closed Dialectica categories. *Annals of pure and applied logic*, 156(2-3):290–307, 2008.

6 Andreas Blass. A game semantics for linear logic. *Annals of pure and applied logic*, 1991.

7 Valeria de Paiva. The dialectica categories. In *Proc. of categories in computer science*, 1989.

8 Valeria de Paiva. Categorical multirelations, linear logic and petri nets. Technical report, University of Cambridge, 1991.

9 Valeria de Paiva. The dialectica categories. Technical report, University of Cambridge, 1991.

10 Valeria de Paiva. Lineales: algebras and categories in the semantics of linear logic. In D. Barker-Plummer, D. Beaver, Johan van Benthem, and P. Scotto di Luzio, editors, *Words, Proofs and Diagrams*. CSLI, 2002.

11 Valeria de Paiva. Dialectica and Chu construtions: Cousins? *Theory and applications of categories*, 17(7):127–152, 2007.

12 Pieter Hofstra. The dialectica monad and its cousins. In *Models, Logics, and Higher-dimensional Categories: A Tribute to the Work of Mihaly Makkai*, volume 53 of *CRM Proceedings and Lecture Notes*, pages 107–139. American Mathematical Society, 2011.

13 Martin Hyland. Proof theory in the abstract. *Annals of pure and applied logic*, 114(1-3):43–78, 2002.

14 Martin Hyland. Slides of an invited lecture 'Fibrations in Logic' at Category Theory 2007, Coimbra, Portrugal. Available electronically at `https://www.dpmms.cam.ac.uk/ martin/Research/Slides/ct2007.pdf`, 2007.

15 Martin Hyland and Luke Ong. Fair games and full completeness for multiplicative linear logic without the MIX rule. Unpublished manuscript, 1993.

16 Martin Hyland and Andrea Schalk. Glueing and orthogonality for models of linear logic. *Theoretical computer science*, 294(1-2):183–231, 2003.

17 Ulrich Kohlenbach. *Applied proof theory: proof interpretations and their use in mathematics*. Springer, 2008.

18 Saunders Mac Lane. *Categories for the working mathematician*. Springer, 1978.

19 Maria Emilia Maietti, Paola Maneggia, Valeria de Paiva, and Eike Ritter. Relating categorical semantics for intuitionistic linear logic. *Applied categorical structures*, 13(1):1–36, 2005.

20 Paul-André Melliès. Asynchronous games 3: An innocent model of linear logic. *Proceedings of the 10th Conference on Category Theory and Computer Science*, 2004.

21 Paul-André Melliès. Asynchronous games 4: A fully complete model of propositional linear logic. *Proceedings of the 20th Conference on Logic in Computer Science*, 2005.

22 Paul-André Melliès. Categorical semantics of linear logic. In *Interactive models of computation and program behaviour*. Société Mathématique de France, 2009.

23 Alexandre Miquel. A model for impredicative type systems with universes, intersection types and subtyping. In *In Proceedings of the 15th Annual IEEE Symposium on Logic in Computer Science (LICS'00)*, 2000.

24 Paulo Oliva. Computational interpretations of classical linear logic. *Proceedings of WoLLIC'07*, 4576:285–296, 2007.

25 Paulo Oliva. An analysis of Gödel's dialectica interpretation via linear logic. *Dialectica*, 62:269–290, 2008.

26 Paulo Oliva. Functional interpretations of linear and intuitionistic logic. *Information and Computation*, 208(5):565–577, 2010.

27 Philip J. Scott. The "Dialectica" interpretation and categories. *Mathematical logic quarterly*, 24(31-36):553–575, 1978.

28 Masaru Shirahata. The dialectica interpretation of first-order classical affine logic. *Theory and applications of categories*, 2006.

29 Matthijs Vákár. Syntax and semantics of linear dependent types. Technical report, University of Oxford, 2014.

The General Universal Property of the Propositional Truncation*

Nicolai Kraus

University of Nottingham
Nottingham, UK
nicolai.kraus@nottingham.ac.uk

Abstract

In a type-theoretic fibration category in the sense of Shulman (representing a dependent type theory with at least **1**, Σ, Π, and identity types), we define the type of *coherently constant* functions $A \xrightarrow{\omega} B$. This involves an infinite tower of coherence conditions, and we therefore need the category to have Reedy limits of diagrams over ω^{op}. Our main result is that, if the category further has propositional truncations and satisfies function extensionality, the type of coherently constant function is equivalent to the type $\|A\| \to B$.

If B is an n-type for a given finite n, the tower of coherence conditions becomes finite and the requirement of nontrivial Reedy limits vanishes. The whole construction can then be carried out in (standard syntactical) homotopy type theory and generalises the universal property of the truncation. This provides a way to define functions $\|A\| \to B$ if B is not known to be propositional, and it streamlines the common approach of finding a propositional type Q with $A \to Q$ and $Q \to B$.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases coherence conditions, propositional truncation, Reedy limits, universal property, well-behaved constancy

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.111

1 Introduction

In homotopy type theory (HoTT), we can *truncate* (*propositionally* or *(-1)-truncate*, to be precise) a type A to get a type $\|A\|$ witnessing that A is inhabited without revealing an inhabitant [27, Chapter 3.7]. This operation roughly corresponds to the *bracket types* [4] of extensional Martin-Löf Type Theory, and to the *squash types* [7] of NuPRL.

The type $\|A\|$ is always propositional, meaning that any two of its inhabitants are equal, and its universal property states that functions $\|A\| \to B$ correspond to functions $A \to B$, provided that B is propositional. In particular, we always have a canonical map $|-|_A : A \to \|A\|$. This definition is natural and elegant, essentially making the truncation operation a reflector of the subcategory of propositions. Unfortunately, it can be rather tricky to define a function $\|A\| \to B$ if B is not known to be propositional.

One possible way to understand the propositional truncation is to think of elements of $\|A\|$ as *anonymous* inhabitants of A, with the function $|-|_A$ hiding the information which concrete element of A one actually has. With this in mind, let us have a closer look at the mentioned universal property of the propositional truncation, or equivalently, at its

* This work was supported by the Engineering and Physical Sciences Research Council (EPSRC), grant reference EP/M016994/1.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 111–145

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

elimination principles. If we want to find an inhabitant of $\|A\| \to B$ and B is propositional, then a function $f : A \to B$ is enough. A possible interpretation of this fact is that f cannot take different values for different inputs, because B is propositional, justifying that f does (in a certain sense) not have to "look at" its argument, such that an anonymous argument is enough. Note that, we only think of internal properties here. When it comes to computation, the *term* f can certainly behave differently if applied on different *terms* of type A.

This thought suggests that, in order to construct and inhabitant of $\|A\| \to B$ if B is not necessarily propositional, we need to put a condition on the function f to make sure that it does not distinguish between different inputs. In other words, we expect that f is required to satisfy some form of *constancy*. The obvious first try would be to ask for an inhabitant of

$$\mathsf{const}_f :\equiv \Pi_{a^1 a^2 : A} f(a^1) = f(a^2), \tag{1}$$

where we write $=$ for the identity type as it has become standard in HoTT. The assumption (1) suffices to derive a function $\|A\| \to B$ if we in addition know that B is a set (also called *h-set*, or said to have *unique identity proofs*). As a central concept of HoTT is that the identity type is not always propositional, it is not surprising that (1) generally only solves the problem if this additional requirement on B is fulfilled. If we have a proof that two elements of a type are equal, it will very often matter in which way they are equal. Thus, the naive statement that f maps any two points to equal values is usually too weak to construct a map out of the propositional truncation. This problem has been studied before [14, 13].

Given a function $f : A \to B$ and a proof $c : \mathsf{const}_f$ of weak constancy, we can ask whether the paths (identity proofs) that c gives are well-behaved in the sense that they fit together. Essentially, if we use c to construct two inhabitants of $f(a_1) = f(a_2)$, then those inhabitants should be equal. If we know this, we can weaken the condition that B is a set to the condition that B is a *groupoid* (i.e. 1-truncated), and still construct a function $\|A\| \to B$. This, and the (simpler) case that B is a set as described above, are presented as Propositions 2 and 3 in Section 2. In principle, we could go on and prove the corresponding statement for the case that B is 2, 3, ...-truncated, each step requiring one additional coherence assumption. Unfortunately, handling long sequences of coherence conditions in the direct syntactic way becomes rather unpleasant very quickly.

A setting in which we can deal nicely with such towers of conditions was given by Shulman [24], who makes precise the idea that type-theoretic contexts (or "nested Σ-types") correspond to diagrams over inverse categories of a certain shape. Although we do not require the main result (the construction of univalent models and several applications) of [24], we make use of the framework and technical results. Working in a type-theoretic fibration category in the sense of Shulman, we can further consider the case that this category has Reedy ω^{op}-limits, that is, limits of infinite sequences $A_1 \twoheadleftarrow A_2 \twoheadleftarrow A_3 \twoheadleftarrow \ldots$, where every map is a fibration (projection). We can think of those limits as "infinite contexts" or "Σ-types with infinitely many Σ-components". If these Reedy limits exist, we can formulate the type of *coherently constant* functions from A to B, for which we write $A \xrightarrow{\omega} B$. We show that such a coherently constant function allows us to define a function $\|A\| \to B$, even if B is not known to be n-truncated for any finite n. Even stronger, the type $A \xrightarrow{\omega} B$ is homotopy equivalent to the type $\|A\| \to B$, in the same way as $A \to B$ is equivalent to $\|A\| \to B$ under the very strict assumption that B is propositional.

The syntactical version of HoTT as presented in the standard reference [27, Appendix A.2] does not have (or is at least not expected to have) Reedy ω^{op}-limits. However, if we consider an n-truncated type B for some finite fixed number n, then all but finitely many of the coherence conditions captured by $A \xrightarrow{\omega} B$ become trivial, and that type can be

simplified to a finitely nested Σ-type for which we will write $A \xrightarrow{[n+1]} B$. It can be formulated in the syntax of HoTT where we can then prove that, for any A and any n-truncated B, the type $A \xrightarrow{[n+1]} B$ is equivalent to $\|A\| \to B$. We thereby generalise the usual universal property of the propositional truncation [27, Lemma 7.3.3], because if B is not only n-truncated, but propositional, then $A \xrightarrow{[n+1]} B$ can be reduced to $A \to B$ simply by removing contractible Σ-components. From the point of view of the standard syntactical version of HoTT, an application of our construction is therefore be the construction of functions $\|A\| \to B$ for the case that B is not propositional. The usual approach for this problem is to construct a propositional type Q such that $A \to Q$ and $Q \to B$ (see [27, Chapter 3.9]). Our construction can be seen as a uniform construction of such a Q, since the equivalence $(A \xrightarrow{[n+1]} B) \simeq (\|A\| \to B)$ is proved by constructing a suitable "contractible extension" of $A \xrightarrow{[n+1]} B$; the general strategy is to "expand and contract" type-theoretic expressions, as we strive to explain with the help of the examples in Section 2.

Nevertheless, we want to stress that we consider the correspondence between $A \xrightarrow{\omega} B$ and $\|A\| \to B$ in a type-theoretic fibration category with Reedy ω^{op}-limits our main result, and the finite special cases described in the previous paragraph essentially fall out as a corollary. In fact, we think that Reedy ω^{op}-limits are a somewhat reasonable assumption. Recently, it has been discussed regularly how these or similar concepts can be introduced into syntactical type theory (for example, see the blog posts by Shulman [23] and Oliveri [20] with the comments sections, and the discussion on the HoTT mailinglist [26] titled "Infinitary type theory"). Motivations are the question whether HoTT can serve as its own meta-theory, whether we can write an interpreter for HoTT in HoTT, and related questions problems such as the definition of semi-simplicial types [9]. Moreover, a concept that is somewhat similar has been suggested earlier as "very dependent types" [10], even though this suggestion was made in the setting of NuPRL [7].

As one anonymous reviewer has pointed out, our main result (Theorem 27) can be seen as a type-theoretic, constructive version of Proposition 6.2.3.4 in Lurie's *Higher Topos Theory* [18]. This seems to suggest once more that many connections between type theory and homotopy and topos theory are unexplored until now. The current author has yet to understand the results by Lurie and the precise relationship.

Contents. We first discuss the cases that the codomain B is a set or a groupoid, as described in the introduction, in Section 2. This provides some intuition for our general strategy of proving a correspondence between coherently constant functions and maps out of propositional truncations. In particular, we describe how the method of "adding and removing contractible Σ-components" for proving equivalences can be applied. In Section 3, we briefly review the notion of a type-theoretic fibration category, of an inverse category, and, most importantly, constructions related to Reedy fibrant diagrams, as described by Shulman [24]. Some simple observations about the restriction of diagrams to subsets of the index categories are recorded in Section 4. We proceed by defining the *equality diagram* over a given type for a given inverse category in Section 5. The special case where the inverse category is Δ_+^{op} (the category of nonempty finite sets and strictly increasing functions) gives rise to the *equality semi-simplicial type*, which is discussed in Section 6. We show that the projection of a full n-dimensional tetrahedron to any of its horns is a homotopy equivalence. Then, in Section 7, we construct a fibrant diagram that represents the exponential of a fibrant and a non-fibrant diagram, with the limit taken at each level. We extend the category Δ_+^{op} in Section 8, which allows us to make precise how contractible Σ-components can be "added

and removed" in general. Our main result, namely that the types $A \xrightarrow{\omega} B$ and $\|A\| \to B$ are homotopy equivalent, is shown in Section 9. The finite special cases which can be done without the assumption of Reedy ω^{op}-limits are proved in Section 10, while Section 11 is reserved for concluding remarks.

Notation. We use type-theoretic notation and we assume familiarity with HoTT, in particular with the book [27] and its terminology. If A is a type and B depends on A, it is standard to write $\Pi_{a:A}B(a)$ or $\Pi_A B$ for the type of dependent functions. For the dependent pair type, we write $\Sigma\,(a : A)\,.\,B(a)$. The reason for this apparent mismatch is that we sometimes have to consider nested Σ-types, and it would seem unreasonable to write all Σ-components apart from the very last one as subscripts. It is sometimes useful to give the last component of a (nested) Σ-type a name, in which case we allow ourselves to write expressions like $\Sigma\,(a : A)\,.\,\Sigma\,(b : B(a))\,.\,(c : C(a, b))$.

Regarding notation, one potentially dangerous issue is that there are many different notions of equality-like concepts, such as the identity type of type theory, internal equivalence of types, judgmental equality of type-theoretic expressions, isomorphism of objects in a category, isomorphism or equivalence of categories, and strict equality of morphisms. For this article, we use the convention that *internal* concepts are written using "two-line" symbols, coinciding with the notation of [27]: we write $a = b$ for the identity type $\mathsf{Id}(a, b)$, and $A \simeq B$ for the type of equivalences between A and B. Other concepts are denoted (if at all) using "three-line" symbols: we write $a \equiv b$ if a and b denote two judgmentally equal expressions, and we use $\equiv$ for other cases of *strict* equality in the meta-theory. By writing $x \cong y$, we express that x and y are isomorphic objects of a category. Equality of morphisms (of a category) is sometimes expressed with $\equiv$, but usually by saying that some diagram commutes, and if we say that some diagram commutes, we always mean that it commutes *strictly*, not only up to homotopy. Other notions of equality are written out.

If C is some category and $x \in C$ an object, we write (as it is standard) x/C for the co-slice category of arrows $x \to y$. We do many constructions involving subcategories, but we want to stress that we always and exclusively work with *full* subcategories (apart from the subcategory of fibrations in Definition 4). Thus, we write $C - x$ for the full subcategory of C that we get by removing the object x. Further, if D is a full subcategory of C (we write $D \subset C$) which does not contain x, we write $D + x$ for the full subcategory of C that has all the objects of D and the object x.

Not exactly notation, but in a similar direction, are the following two remarks: First, when we refer to the *distributivity law* of Π and Σ, we mean the equivalence

$$\Pi_{a:A}\Sigma\,(b : B(a))\,.\,C(a, b) \;\simeq\; \Sigma\,(f : \Pi_{a:A}B(a))\,.\,\Pi_{a:A}C(a, f(a)) \tag{2}$$

which is sometimes referred to as the *type-theoretic axiom of choice* or AC_∞ (see [27]). Second, if we talk about a *singleton*, we mean a type expression of the form $\Sigma\,(a : A)\,.\,a = x$ or $\Sigma\,(a : A)\,.\,x = a$ for a fixed x. The term *singleton* therefore refers to a syntactical shape in which some types can be represented, and it is well-known that those types are contractible.

2 A First Few Special Cases

In this section, we want to discuss some simple examples and aim to build up intuition for the general case. For now, we work entirely in standard (syntactical) homotopy type theory as specified in [27, Appendix A.2], together with function extensionality (see [27, Appendix A.3.1]) and propositional truncation. To clarify the latter, we assume that, for any type A,

there is a propositional type $\|A\|$ with a function $|-|_A : A \to \|A\|$. Composition with $|-|_A$ is moreover assumed to induce an equivalence $(\|A\| \to B) \simeq (A \to B)$. Due to the "equivalence reasoning style" nature of our proofs, we can avoid the necessity of any "unpleasant manual computation". Thus, we would not benefit from the judgmental computation rule that is usually imposed on the propositional truncation (other than not having to assume function extensionality explicitly [14]). We think it is worth mentioning that we actually do not require much of the power of homotopy type theory: we only use 1, Σ, Π, identity types, propositional truncations, and assume function extensionality. This will in later sections turn out to be a key feature which enables us to perform the construction in the infinite case (assuming the existence of certain Reedy limits).

Assume we want to construct an inhabitant of $\|A\| \to B$ and B is an n-type, for a fixed given n. The case $n \equiv -2$ is trivial. For $n \equiv -1$, the universal property (or the elimination principle) can be applied directly. In this section, we explain the cases $n \equiv 0$ and $n \equiv 1$. The following auxiliary statement will be useful:

▶ **Lemma 1.** *Let $C_1, C_2, \dots, C_m$ be types dependent on A, possibly with C_j depending on C_i for $i < j$. Consider a nested Σ-type, built out of Σ-components of the form $\Pi_A C_k$. Then, functions from $\|A\|$ into that type correspond directly to elements of that type. That is, the types*

$$\begin{aligned}\|A\| \to \big(&\Sigma\,(f_1 : \Pi_{a:A} C_1(a))\,.\\ &\Sigma\,(f_2 : \Pi_{a:A} C_2(a, f_1(a)))\,.\\ &\Sigma \quad \dots\\ &\quad (\Pi_{a:A} C_m(a, f_1(a), f_2(a), \dots, f_{m-1}(a)))\big)\end{aligned} \tag{3}$$

and

$$\begin{aligned}&\Sigma\,(f_1 : \Pi_{a:A} C_1(a))\,.\\ &\Sigma\,(f_2 : \Pi_{a:A} C_2(a, f_1(a)))\,.\\ &\Sigma \dots\\ &\quad (\Pi_{a:A} C_m(a, f_1(a), f_2(a), \dots, f_{m-1}(a)))\end{aligned} \tag{4}$$

are equivalent.

Proof. This holds by the usual distributivity law (2) of Π (or $\to$) and Σ, together with the equivalence $\|A\| \times A \simeq A$. ◀

2.1 Constant Functions into Sets

We consider the case $n \equiv 0$ first; that is, we assume that B is a set. Recall the definition of const given in (1).

▶ **Proposition 2** ($\mathsf{case}\ n \equiv 0$)**.** *Let B be a set and A any type. Then, we have the equivalence*

$$(\|A\| \to B) \;\simeq\; \Sigma\,(f : A \to B)\,.\,\mathsf{const}_f. \tag{5}$$

Note that, if B is not only a set but even a propositional type, the condition const_f is not only automatically satisfied, but it is actually contractible as a type. By the usual equivalence lemmata, the type on the right-hand side of (5) then simplifies to $(A \to B)$, which exactly is the universal property. Thus, we view (5) as a first generalisation.

Proof of Proposition 2. Assume $\mathfrak{a}_\circ : A$ is some point in A. In the following, we construct a chain of equivalences. The variable names for certain Σ-components might seem somewhat odd: for example, we introduce a point $f_1 : B$. The reason for this choice will become clear later. For now, we simply emphasise that f_1 is "on the same level" as $f : A \to B$ in the sense that they both give points, rather than for example paths (like, for example, an inhabitant of const_f).

$$\begin{aligned}
& B \\
(\text{S1}) \quad &\simeq \Sigma\,(f_1 : B)\,.\,\big(A \to \Sigma\,(b : B)\,.\,b = f_1\big) \\
(\text{S2}) \quad &\simeq \Sigma\,(f_1 : B)\,.\,\Sigma\,(f : A \to B)\,.\,\Pi_{a:A} f(a) = f_1 \\
(\text{S3}) \quad &\simeq \Sigma\,(f_1 : B)\,.\,\Sigma\,(f : A \to B)\,.\,(\Pi_{a:A} f(a) = f_1) \times (\mathsf{const}_f) \times (f(\mathfrak{a}_\circ) = f_1) \qquad (6) \\
(\text{S4}) \quad &\simeq \Sigma\,(f : A \to B)\,.\,(\mathsf{const}_f) \times \Sigma\,(f_1 : B)\,.\,(f(\mathfrak{a}_\circ) = f_1) \times (\Pi_{a:A} f(a) = f_1) \\
(\text{S5}) \quad &\simeq \Sigma\,(f : A \to B)\,.\,(\mathsf{const}_f) \times (\Sigma\,(f_1 : B)\,.\,f(\mathfrak{a}_\circ) = f_1) \\
(\text{S6}) \quad &\simeq \Sigma\,(f : A \to B)\,.\,\mathsf{const}_f
\end{aligned}$$

Let us explain the validity of the single steps. In the first step, we add a family of singletons. In the second step, we apply the distributivity law (2). In the third step, we add two Σ-components, and B being a set ensures that both of them are propositional. But it is very easy to derive both of them from $\Pi_{a:A} f(a) = f_1$, showing that both of them are contractible. In the fourth step, we simply reorder some Σ-components, and in the fifth step, we use that $\Pi_{a:A} f(a) = f_1$ is contractible by an argument analogous to that of the third step. Finally, we can remove two Σ-components which form a contractible singleton.

If we carefully trace the equivalences, we see that the function part

$$e : B \;\to\; \Sigma\,(f : A \to B)\,.\,\mathsf{const}_f \qquad (7)$$

is given by

$$e(b) \equiv \big(\lambda a.b\,,\; \lambda a^1 a^2.\mathsf{refl}_b\big)\,, \qquad (8)$$

not depending on the assumed $\mathfrak{a}_\circ : A$. But as e is an equivalence assuming A, it is also an equivalence assuming $\|A\|$.

As $\|A\| \to \big(B \simeq (\Sigma\,(f : A \to B)\,.\,\mathsf{const}_f)\big)$ implies that the two types $(\|A\| \to B)$ and $\big(\|A\| \to (\Sigma\,(f : A \to B)\,.\,\mathsf{const}_f)\big)$ are equivalent, the statement follows from Lemma 1. ◀

The core strategy of the steps (S1) to (S6) is to add and remove contractible Σ-components, and to reorder and regroup them. This principle of expanding and contracting a type expression can be generalised and, as we will see, even works for the infinite case when B is not known to be of any finite truncation level. Generally speaking, we use two ways of showing that components of Σ-types are contractible. The first is to group two of them together such that they form a singleton, as we did in (S1) and (S6). The second is to use the fact that B is truncated, as we did in (S3). We consider the first to be the key technique, and in the general (infinite) case of an untruncated B, the second can not be applied at all. We thus view the second method as a tool to deal with single Σ-components that lack a "partner" only because the case that we consider is finite, and which is unneeded in the infinite case.

2.2 Constant Functions into Groupoids

The next special case is $n \equiv 1$. Assume that B is a 1-type (sometimes called a *groupoid*). Let us first clarify which kind of constancy we expect for a map $f : A \to B$ to be necessary.

Not only do we require $c : \mathsf{const}_f$, we also want this constancy proof (which is in general not propositional any more) to be *coherent*: given a^1 and $a^2 : A$, we expect that c only allows us to construct essentially *one* proof of $f(a^1) = f(a^2)$. The reason is that we want the data (which includes f and c) together to be just as powerful as a map $\|A\| \to B$, and from such a map, we only get trivial loops in B.

We claim that the required coherence condition is

$$\mathsf{coh}_{f,c} :\equiv \Pi_{a^1 a^2 a^3 : A} c(a^1, a^2) \cdot c(a^2, a^3) = c(a^1, a^3). \tag{9}$$

A first sanity check is to see whether from $d : \mathsf{coh}_{f,c}$ we can now prove that $c(a, a)$ is equal to refl_a, something that should definitely be the case if we do not want to be able to construct possibly different parallel paths in B. To give a positive answer, we only need to see what $d(a, a, a)$ tells us.

▶ **Proposition 3** (case $n \equiv 1$)**.** *Let B be a groupoid (1-type) and A be any type. Then, we have*

$$(\|A\| \to B) \;\simeq\; \big(\Sigma\,(f : A \to B)\,.\,\Sigma\,(c : \mathsf{const}_f)\,.\,\mathsf{coh}_{f,c}\big). \tag{10}$$

Note that Proposition 3 generalises Proposition 2: if B is a set (as in Proposition 2), it is also a groupoid and the type $\mathsf{coh}_{f,c}$ becomes contractible, as it talks about equality of equalities.

Proof. Although not conceptually harder, it is already significantly more tedious to write down the chain of equivalences. We therefore choose a slightly different representation. Assume $\mathfrak{a}_0 : A$ as before. We then have:

$$\begin{aligned}
& B \\
\text{(S1)}\quad \simeq\; & \\
& \Sigma\,(f_1 : B)\,. \\
& \Sigma\,(f : A \to B)\,.\,\Sigma\,(c_1 : \Pi_{a:A} f(a) = f_1)\,. \\
& \Sigma\,(c : \mathsf{const}_f)\,.\,\Sigma\,\big(d_1 : \Pi_{a^1 a^2 : A} c(a^1, a^2) \cdot c_1(a^2) = c_1(a^1)\big)\,. \\
& \Sigma\,(c_2 : f(\mathfrak{a}_0) = f_1)\,.\,\Sigma\,(d_3 : c(\mathfrak{a}_0, \mathfrak{a}_0) \cdot c_1(\mathfrak{a}_0) = c_2)\,. \\
& \Sigma\,(d : \mathsf{coh}_{f,c_0})\,. \\
& \quad (d_2 : \Pi_{a:A} c(\mathfrak{a}_0, a) \cdot c_1(a) = c_2) \\
\text{(S2)}\quad \simeq\; & \\
& \Sigma\,(f : A \to B)\,.\,\Sigma\,(c : \mathsf{const}_f)\,.\,\Sigma\,(d : \mathsf{coh}_{f,c})\,. \\
& \Sigma\,(f_1 : B)\,.\,\Sigma\,(c_2 : f(\mathfrak{a}_0) = f_1)\,. \\
& \Sigma\,(c_1 : \Pi_{a:A} f(a) = f_1)\,.\,\Sigma\,(d_2 : \Pi_{a:A} c(\mathfrak{a}_0, a) \cdot c_1(a) = c_2)\,. \\
& \Sigma\,\big(d_1 : \Pi_{a^1 a^2 : A} c(a^1, a^2) \cdot c_1(a^2) = c_1(a^1)\big)\,. \\
& \quad (d_3 : c(\mathfrak{a}_0, \mathfrak{a}_0) \cdot c_1(\mathfrak{a}_0) = c_2) \\
\text{(S3)}\quad \simeq\; & \\
& \Sigma\,(f : A \to B)\,.\,\Sigma\,(c : \mathsf{const}_f)\,.\,(d : \mathsf{coh}_{f,c})
\end{aligned} \tag{11}$$

In the first step (S1), we expand the single type B to a nested Σ-type with in total nine Σ-components. We write them in six lines, and each line apart from the first is a contractible part of this nested Σ-type, implying that the whole type is equivalent to B. In the lines two and three, we can apply the distributivity law, i.e. the equivalence (2), to give them

the shape of singletons, while the fourth line is already a singleton. As B is 1-truncated, the lines five and six represent propositional types, but those types are easily seen to be inhabited using the other Σ-components.

In the second step, we simply re-order some Σ-components. Then, in step (S3), we remove the Σ-components in the lines two to five which is justified as, again, each line represents a contractible part of the nested Σ-type.

We trace the canonical equivalences to see that the function-part of the constructed equivalence is

$$e : B \to \Sigma\,(f : A \to B)\,.\,\Sigma\,(c : \mathsf{const}_f)\,.\,(d : \mathsf{coh}_{f,c}) \tag{12}$$

$$e(b) \equiv (\lambda a.b\,,\ \lambda a^1 a^2.\mathsf{refl}_b\,,\ \lambda a^1 a^2 a^3.\mathsf{refl}_{\mathsf{refl}_b}). \tag{13}$$

In particular, e is independent from the assumed $\mathfrak{a}_\circ : A$. As before, this means that e is an equivalence assuming $\|A\|$, and, with the help of Lemma 1, we derive the claimed equivalence. ◀

2.3 Outline of the General Idea

At this point, it seems plausible that what we have done for the special cases of $n \equiv 0$ and $n \equiv 1$ can be done for any (fixed) $n < \infty$. Nevertheless, we have seen that the case of groupoids is already significantly more involved than the case of sets. To prove a generalisation, we have to be able to state what it means for a function to be "coherently constant" on n levels, rather than just the first one or two.

Let us try to specify what "coherently constant" should mean in general. If we have a function $f : A \to B$, we get a point in B for any $a : A$. A constancy proof $c : \mathsf{const}_f$ gives us, for any pair of points in A, a path between the corresponding points in B. Given three points, c gives us three paths which form a "triangle", and an inhabitant of $\mathsf{coh}_{f,c}$ does nothing else than providing a filler for such a triangle. It does not take much imagination to assume that, on the next level, the appropriate coherence condition should state that the "boundary" of a tetrahedron, consisting of four filled triangles, can be filled.

To gain some intuition, let us look at the following diagram:

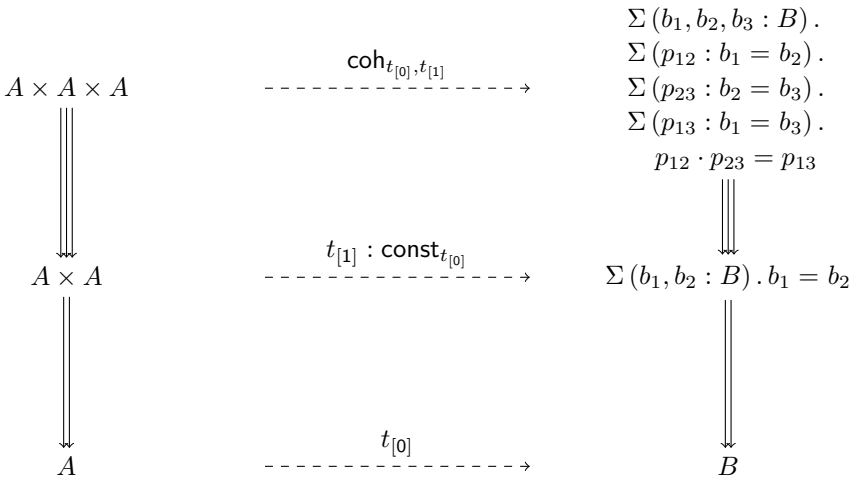


Figure 1 Constancy as a natural transformation.

All vertical arrows are given by projections. Consider the category D with objects the finite ordinals $[0]$, $[1]$ and $[2]$ (with 1, 2, and 3 objects, respectively), and arrows the strictly monotonous maps. Then, the left-hand side and the right-hand side can both be seen as a diagram over D^{op}. The data that we need for a "coherently constant function" from A into B, if B is a groupoid, can now be viewed as a natural transformation t from the left to the right diagram (the dashed horizontal arrows). On the lowest level, such a natural transformation consists of a function $t_{[0]} : A \to B$, which we called f before. On the next level, we have $t_{[1]} : A^2 \to \Sigma\,(b_1, b_2 : B)\,.\,b_1 = b_2$, but in such a way that the diagram commutes (strictly, not up to homotopy), enforcing

$$\mathsf{fst}(t_{[1]}(a^1, a^2)) \equiv (t_{[0]}(a^1), t_{[0]}(a^2)) \tag{14}$$

and thereby making $t_{[1]}$ the condition that $t_{[0]}$ is weakly constant. Finally, $t_{[2]}$ yields the coherence condition coh.

In the most general case, where we do not put any restriction on B, we certainly cannot expect that a finite number of coherence conditions can suffice. Instead of the diagram over D^{op}, as pictured on the right-hand side of Figure 1, we will need a diagram over the the category of all non-zero finite ordinals. This is what we call the *equality semi-simplicial type* over B, written $\mathcal{E}B$. In the language of model categories, this is a fibrant replacement of the constant diagram. It would be reasonable to expect that our $\mathcal{E}B$ extends the diagram shown in Figure 1, but this will only be true up to (levelwise) equivalence of types. Defining $\mathcal{E}B$ as a strict extension of that diagram is tempting, but it seems to be combinatorically nontrivial to continue in the same style, as it would basically need Street's orientals [25]. Our construction will be much simpler to write down and easier to work with, with the only potential disadvantage being that, compared to the diagram Figure 1, the lower levels will look rather bloated. The other diagram in Figure 1, i.e. the left-hand side, is easy to extend, and we call it the *trivial* diagram over A. In the terminology of simplicial sets, it is the $[0]$-coskeleton of the constant diagram. Our main result is essentially an internalised version, stated as an equivalence of types, of the following slogan:

> *Functions $\|A\| \to B$ correspond to natural transformations from the trivial diagram over A to the semi-simplicial equality type over B.*

Our type of natural transformations is basically a Reedy limit of an exponential of diagrams. We will perform the *expanding and contracting* principle that we have exemplified in the proofs of Propositions 2 and 3 by modifying the index category of the diagram of which we take the limit, step by step, taking care that every single step preservers the Reedy limit in question up to homotopy equivalence. As we will see, these steps correspond indeed to the steps that we took in the proofs of Propositions 2 and 3.

3 Fibration Categories, Inverse Diagrams, and Reedy Limits

In his work on *Univalence for Inverse Diagrams and Homotopy Canonicity*, Shulman has proved several deep results [24]. Among other things, he shows that diagrams over inverse categories can be used to build new models of univalent type theory, and uses this to prove a partial solution to Voevodsky's homotopy-canonicity conjecture. We do not require those main results; in fact, we do not even assume that there is a universe, and consequently we also do not use univalence! At the same time, what we want to do can be explained nicely in terms of diagrams over inverse diagrams, and we therefore choose to work in the same setting. Luckily, it is possible to do this with only a very short introduction to type-theoretic

fibration categories, inverse diagrams and Reedy limits, and this is what the current section servers for.

Type-theoretic fibration categories. A *type-theoretic fibration category* (as defined in [24, Definition 2.1] is a category with some structure that allows to model dependent type theory with identity types. Let us recall the definition, where we use a lemma by Shulman to give an equivalent (more "type-theoretic") formulation:

▶ **Definition 4** (Type-theoretic fibration category, [24, Definition 2.1 combined with Lemma 2.4])**.** A type-theoretic fibration category is a category $\mathfrak{C}$ which has the following structure.

(i) A terminal object $\mathbf{1}$.

(ii) A (not necessarily full) subcategory $\mathfrak{F} \subset \mathfrak{C}$ containing all the objects, all the isomorphisms, and all the morphisms with codomain $\mathbf{1}$. A morphism in $\mathfrak{F}$ is called a *fibration*, and written as $A \twoheadrightarrow B$. Any morphism i is called an *acyclic cofibration* and written $i : X \xrightarrow{\sim} Y$ if it has the left lifting property with respect to all fibrations, meaning that every commutative square

$$\begin{array}{ccc} X & \longrightarrow & A \\ {\scriptstyle i}\downarrow{\scriptstyle\sim} & & \downarrow{\scriptstyle f} \\ Y & \longrightarrow & B \end{array}$$

has a (not necessarily unique) filler $h : Y \to A$ that makes both triangles commute.

(iii) All pullbacks of fibrations exist and are fibrations.

(iv) For every fibration $g : A \twoheadrightarrow B$, the pullback functor $g^\star : \mathfrak{C}/B \to \mathfrak{C}/A$ has a partial right adjoint Π_g, defined at all fibrations over A, whose values are fibrations over B.

(v) For any fibration $A \twoheadrightarrow B$, the diagonal $A \to A \times_B A$ factors as $A \xrightarrow{\sim} P_B A \twoheadrightarrow A \times_B A$, with the first map being an acyclic cofibration and the second being a fibration.

(vi) For any $A \twoheadrightarrow B$, there exists a factorisation as in (5) such that in any diagram of the shape

$$\begin{array}{ccccc} X & \longrightarrow & Y & \longrightarrow & Z \\ \downarrow & & \downarrow & & \downarrow \\ A & \xrightarrow{\sim} & P_B A & \twoheadrightarrow & B \end{array}$$

we have the following: if both squares are pullback squares (which implies that $Y \to Z$ and $X \to Z$ are fibrations), then $X \to Y$ is an acyclic cofibration.

▶ Remark. From the above definition, it follows that *every* morphism factors as an acyclic cofibration followed by a fibration. Shulman's proof [24, Lemma 2.4], a translation of the proof by Gambino and Garner [8] into category theory, relies on the fact that every morphism $A \to \mathbf{1}$ is a fibration ("all objects are fibrant") by definition.

The example of a type-theoretic fibration category that we mainly have in mind is [24, Example 2.9], the category of contexts of a dependent type theory with a unit type, Σ- and Π-types, and identity types. The unit, Σ- and Π-types are required to satisfy judgmental η-rules. Because of these η-rules, we do not need to talk about contexts; we can view every object of the category as a nested Σ-type with some finite number of components. Of course, the terminal object is the unit type. The subset of fibrations is the closure of the projections under isomorphisms. One nice property is that the η-rules also imply that we can assume that all fibrations are a projection of the form $\big(\Sigma\,(x : X)\,.\,Y(x)\big) \twoheadrightarrow X$. Pullbacks correspond to substitutions, and the partial functor Π_g comes from dependent function types. For any fibration $f : A \twoheadrightarrow B$, the factorisation in item (5) can be obtained using

the intensional identity type: if B is the unit type, then the factorisation can be written as $A \xrightarrow{\sim} \left(\Sigma\left((x,y) : A \times A\right).\, x = y\right) \twoheadrightarrow A \times A$, and similar otherwise (see [8]). The acyclic cofibration is given by reflexivity.

Note that the type theory specified in the standard reference on HoTT [27, Appendix A.2] does not have judgmental η-rules for Σ and $\mathbf{1}$. This does not constitute a problem when we want to apply our results to homotopy type theory. First, it appears to be an arbitrary choice of [27] to not include these judgmental η-rules in the theory. There does not seem to be any fundamental difficulty with them, and the implementations Agda and Coq do indeed support them. Second, as Shulman states, these judgmental η-rules are convenient but not really necessary [24, Example 2.9]. This is certainly true for our constructions that we can do with finitely nested Σ-types, although it is likely that the assumption of ω^{op}-limits (infinitely nested Σ-types) would have to be phrased more carefully in the absence of judgmental η-conversions (see our proof of Theorem 27).

Given a type-theoretic fibration category $\mathfrak{C}$ with an object A, we can think of A as a context. Type theoretically, we can work in the theory over the fixed context A. Categorically, this means we work in the slice over A. The slice category $\mathfrak{C}/A$ is not necessarily a type-theoretic fibration category as not all morphisms $B \to A$ are fibrations, but we can simply restrict ourselves to those that are. Shulman denotes this full subcategory of $\mathfrak{C}/A$ by $(\mathfrak{C}/A)_{\mathsf{f}}$. The observation that the (restricted) slice of a type-theoretic fibration category is again a type-theoretic fibration category allows us that, when we want to do an "internally expressible" construction for any general given fibration, we can without loss of generality assume that the codomain of the fibration is the unit type. This corresponds to the fact that an "internal" construction in type theory still works if we add additional assumptions to the context (which are then simply ignored by the construction).

It is not exactly true that a type-theoretic fibration category has an intensional dependent type theory as its internal language due to the well-known issue that substitution in type theory is strictly functorial. Fortunately, coherence theorems (see e.g. [3, 17]) can be applied to solve this problem, and we do not worry about it but simply refer to Shulman's explanation [24, Chapter 4]. The crux is that, disregarding these coherence issues, the syntactic category of the dependent type theory with $\mathbf{1}$, Σ, Π, and identity types is essentially the initial type-theoretic fibration category. A consequence we will exploit heavily is that, when reasoning about type-theoretic fibration categories, we can use type-theoretic constructions freely as long as they can be performed using $\mathbf{1}$, Π, Σ, and identity types. For example, the same notion of function extensionality and type equivalence $A \simeq B$ can be defined. This means, of course, that we have to be very careful with the terminology. We call a morphism that is an equivalence in the type-theoretic sense a *homotopy equivalence*, written $A \xrightarrow{\sim} B$, while an *isomorphism* is really an isomorphism in the usual categorical sense. Note that any isomorphism is not only a fibration by definition, but it is automatically an acyclic cofibration, and acyclic cofibrations are further automatically homotopy equivalences. Further, it is natural to introduce the following terminology:

▶ **Definition 5** (Acyclic fibration)**.** We say that a morphism is an *acyclic fibration* if it is a fibration and a homotopy equivalence.

An important property to record is that acyclic fibrations are stable under pullback [24, Corollary 3.12]. In diagrams, we write $A \xrightarrow{\sim} B$ for acyclic fibrations.

Inverse categories and Reedy fibrant diagrams. For objects x and y of a category, write $y \prec x$ if y receives a nonidentity morphism from x (and $y \preceq x$ if $y \prec x$ or $y \equiv x$). A category

$\mathfrak{I}$ is called an *inverse category* (also sometimes called *one-way category*) if the relation $\prec$ is well-founded. In this case, the *ordinal rank* of an object x in $\mathfrak{I}$ is defined by

$$\rho(x) :\equiv \sup_{y \prec x}(\rho(y) + 1). \tag{15}$$

As described by Shulman [24, Section 11], diagrams on $\mathfrak{I}$ can be constructed by well-founded induction in the following way. If x is an object, write $x /\!\!/ \mathfrak{I}$ for the full subcategory of the co-slice category $x/\mathfrak{I}$ which excludes only the identity morphism id_x. Consider the full subcategory $\{\, y \mid y \prec x \,\} \subset \mathfrak{I}$. There is the forgetful functor $U : x /\!\!/ \mathfrak{I} \to \{\, y \mid y \prec x \,\}$, mapping any $x \xrightarrow{f} y$ to its codomain y. If further A is a diagram in a type-theoretic fibration category $\mathfrak{C}$ that is defined on this full subcategory, if the limit

$$M_x^A :\equiv \mathsf{lim}_{x /\!\!/ \mathfrak{I}}(A \circ U). \tag{16}$$

exists, it it called the corresponding *matching object*. To extend the diagram A to the full subcategory $\{\, y \mid y \preceq x \,\} \subset \mathfrak{I}$, it is then sufficient to give an object A_x and a morphism $A_x \to M_x^A$. The diagram $A : \mathfrak{I} \to \mathfrak{C}$ is *Reedy fibrant* if all matching objects M_x^A exist and all the maps $A_x \to M_x^A$ are fibrations. We use the fact that fibrations can be regarded as "one-type projections" in the following way:

▶ **Definition 6** (Decomposition in matching object and fibre)**.** If $A : \mathfrak{I} \to \mathfrak{C}$ is a Reedy fibrant diagram, we write (as said above) M_x^A for its matching objects, and $F^A(x, m)$ for the fibre over m; that is, we have

$$A_x \;\cong\; \Sigma\left(m : M_x^A\right). F^A(x, m). \tag{17}$$

There is the more general notion of a *Reedy fibration* (a natural transformation between two diagrams over $\mathfrak{I}$ with certain properties), so that a diagram is Reedy fibrant if and only if the unique transformation to the terminal diagram is a Reedy fibration. Further, $\mathfrak{C}$ is said to have *Reedy* $\mathfrak{I}$*-limits* if any Reedy fibrant $A : \mathfrak{I} \to \mathfrak{C}$ has a limit which behaves in the way one would expect; in particular, if a natural transformation between two Reedy fibrant diagrams is levelwise a homotopy equivalence, then the map between the limits is a homotopy equivalence. We omit the exact definitions as our constructions do not require them and refer to [24, Chapter 11] for the details instead. For us, it is sufficient to record that a consequence of the definition of having Reedy ω^{op}-limits is the following:

▶ **Lemma 7.** *Let a type-theoretic fibration category $\mathfrak{C}$ that has Reedy ω^{op}-limits be given. Suppose that*

$$F :\equiv F_0 \xleftarrow{\sim} F_1 \xleftarrow{\sim} F_2 \xleftarrow{\sim} \dots \tag{18}$$

is a diagram $F : \omega^{\mathrm{op}} \to \mathfrak{C}$, where all maps are acyclic fibrations. For each i, the canonical map $\mathsf{lim}(F) \to F_i$ is a homotopy equivalence.

Proof. Consider the diagram that is constantly F_i apart from a finite part,

$$G := F_0 \xleftarrow{\sim} F_1 \xleftarrow{\sim} \dots \xleftarrow{\sim} F_{i-1} \xleftarrow{\sim} F_i \xleftarrow{\sim} F_i \xleftarrow{\sim} F_i \dots. \tag{19}$$

There is a canonical natural transformation $F \to G$, induced by the arrows in F, which is a Reedy fibration and levelwise an acyclic fibration. It follows directly from the precise definition of Reedy limits [24, Definition 11.4] that the induced map between the limits $\mathsf{lim}(F) \to F_i$ is a fibration and a homotopy equivalence. ◀

For later, we further record the following two simple lemmata:

▶ **Lemma 8.** *If $A : \mathfrak{I} \to \mathfrak{C}$ is Reedy fibrant, then so is $A \circ U : x/\mathfrak{I} \to \mathfrak{C}$.*

Proof. This is due to the fact that for a (nonidentity) morphism $k : x \to y$ in $\mathfrak{I}$ the categories $k /\!\!/ (x /\!\!/ \mathfrak{I})$ and $y /\!\!/ \mathfrak{I}$ are isomorphic. This argument is already used by Shulman ([24, Lemma 11.8]). ◀

▶ **Lemma 9.** *If $\mathfrak{I}$ is a poset (a partially ordered set), x an object, $A : \mathfrak{I} \to \mathfrak{C}$ a diagram, and the limit $\mathsf{lim}_{x/\!\!/\mathfrak{I}}(A \circ U)$ exists, then $\mathsf{lim}_{\{\, y \mid y \prec x \,\}} A$ exists as well and both are isomorphic.* ◀

An inverse category $\mathfrak{I}$ is *admissible* for $\mathfrak{C}$ if $\mathfrak{C}$ has all Reedy $(x /\!\!/ \mathfrak{I})$-limits. If $\mathfrak{I}$ is finite, then any type-theoretic fibration category has Reedy $\mathfrak{I}$-limits by [24, Lemma 11.8]. From the same lemma, it follows that for all constructions that we are going to do, it will be sufficient if $\mathfrak{C}$ has Reedy ω^{op}-limits. Further, in all our cases of interest, all co-slices of $\mathfrak{I}$ are finite, and $\mathfrak{C}$ is automatically admissible.

Because of the above, let us fix the following:

▶ **Convention 10.** For the rest of this article, let $\mathfrak{C}$ be a type-theoretic fibration category with Reedy ω^{op}-limits, which further satisfies function extensionality. We refer to the objects of $\mathfrak{C}$ (which are by definition always fibrant) as *types*. Let us further introduce the term *tame category*. We say that an inverse category is a *tame category* if all co-slices $x/\mathfrak{I}$ are finite (which implies that $\rho(x)$ is finite for all objects x) and, for all n, the set of objects at "level" n, that is $\{\, x \in \mathfrak{I} \mid \rho(x) \equiv n \,\}$, is finite. The important property is that a tame category $\mathfrak{I}$ is admissible for $\mathfrak{C}$, and that $\mathfrak{C}$ has Reedy $\mathfrak{I}$-limits. Thus, tame categories make it possible to perform constructions without worrying whether required limits exist, and we will not be interested in any non-tame inverse categories.

4 Subdiagrams

Let $\mathfrak{I}$ be a tame category. We are interested in full subcategories of $\mathfrak{I}$, and we mean "subcategory" in the strict sense that the set of objects is a subset of the set of objects of $\mathfrak{I}$. We say that a full subcategory J of $\mathfrak{I}$ is downwards closed if, for any pair x, y of objects in $\mathfrak{I}$ with $y \prec x$, if x is in J, then so is y. The full downwards closed subcategories of $\mathfrak{I}$ always form a poset $\mathsf{Sub}(\mathfrak{I})$, with an arrow $J \to J'$ if J' is a subcategory of J.

It is easy to see that the poset $\mathsf{Sub}(\mathfrak{I})$ has all limits and colimits. For example, given downwards closed full subcategories J and J', their product is given by taking the union of their sets of objects. We therefore write $J \cup J'$. Dually, coproducts are given by intersection and we can write $J \cap J'$. An object x of $\mathfrak{I}$ generates a subcategory $\{\, y \mid y \preceq x \,\}$, for which we write $\overline{x}$.

If $A : \mathfrak{I} \to \mathfrak{C}$ is a Reedy fibrant diagram and $\mathfrak{C}$ has Reedy $\mathfrak{I}$-limits, we can consider the functor

$$\mathsf{lim}_{\_} A : \mathsf{Sub}(\mathfrak{I}) \to \mathfrak{C} \tag{20}$$

which maps any downwards closed full subcategory $J \subseteq \mathfrak{I}$ to $\mathsf{lim}_J A$, the Reedy limit of A restricted to J.

▶ **Lemma 11.** *Let $\mathfrak{I}$ be a tame category and J, K two downwards closed subcategories of $\mathfrak{I}$. Then, the functor $\mathsf{lim}_{\_} A$ maps the pullback square*

$$\begin{array}{ccc} J \cup K & \longrightarrow & K \\ \downarrow & & \downarrow \\ J & \longrightarrow & J \cap K \end{array}$$

in $\mathsf{Sub}(\mathfrak{I})$ *to a pullback square in* $\mathfrak{C}$.

Proof. For an object X, a cone $X \to A|_{J \cup K}$ corresponds to a pair of two cones, $X \to A|_J$ and $X \to A|_K$, which coincide on $J \cap K$. ◀

▶ **Lemma 12.** *Under the same assumptions as before, the functor* lim_A *maps all morphisms to fibrations. In other word, if* K *is a downwards closed subcategory of the inverse category* J*, then*

$$\mathsf{lim}_J A \twoheadrightarrow \mathsf{lim}_K A \tag{21}$$

is a fibration.

Proof. We only need to consider the case that J has exactly one object that K does not have, say $J \equiv K + x$, because the composition of fibrations is a fibration (this is true even for "infinite compositions", with the same short proof as Lemma 7). Further, we may assume that all objects of J are predecessors of x, i.e. we have $\overline{x} \equiv J$; otherwise, we could view $J \to K$ as a pullback of $\overline{x} \to \overline{x} - x$ and apply Lemma 11.

The cone $\mathsf{lim}_K A \to A|_K$ gives rise to a cone $\mathsf{lim}_K A \to (A \circ U)|_{x/\!/K}$ (the morphism into $x \xrightarrow{f} y$ is given by the morphism into y), and we thereby get a morphism $m : \mathsf{lim}_K A \to M_x^A$. If we pull the fibration $A_x \twoheadrightarrow M_x^A$ back along the morphism m, we get a fibration $P \twoheadrightarrow \mathsf{lim}_K A$, and it is easy to see that $P \cong \mathsf{lim}_J A$. ◀

▶ Remark. The above proof gives us a description of the fibration $\mathsf{lim}_{K+x} A \twoheadrightarrow \mathsf{lim}_K A$ in type-theoretic notation. It can be written as

$$\Sigma\,(k : \mathsf{lim}_K A)\,.\,F^A(x, m(k)) \twoheadrightarrow \mathsf{lim}_K A. \tag{22}$$

This remains true even if not all objects in J are predecessors of x.

5 Equality Diagrams

Given any tame category $\mathfrak{I}$ and a fixed type B in $\mathfrak{C}$, the diagram $\mathfrak{I} \to \mathfrak{C}$ that is constantly B is, in general, not Reedy fibrant. Fortunately, the axioms of a type-theoretic fibration category allow us to define a *fibrant replacement* (see, for example, Hoveys textbook [12]). We call the resulting diagram, which we construct explicitly, the *equality diagram* of B over $\mathfrak{I}$. We define by simultaneous induction:

- **(i)** a diagram $\mathcal{E}B : \mathfrak{I} \to \mathfrak{C}$, the *equality diagram*
- **(ii)** a cone $\eta : B \to \mathcal{E}B$ (i.e. a natural transformation from the functor that is constantly B to $\mathcal{E}B$)
- **(iii)** a diagram $M^{\mathcal{E}B} : \mathfrak{I} \to \mathfrak{C}$ (the diagram of matching objects)
- **(iv)** an auxiliary cone $\tilde{\eta} : B \to M^{\mathcal{E}B}$.
- **(v)** a natural transformation $\iota : \mathcal{E}B \to M^{\mathcal{E}B}$

such that $\iota \circ \eta$ equals $\tilde{\eta}$.

Assume that i is an object in $\mathfrak{I}$ such that the five components are defined for all predecessors of i. This is in particular the case if i has no predecessors. We define the matching object $M_i^{\mathcal{E}B} :\equiv \mathsf{lim}_{i/\!/\mathfrak{I}} \mathcal{E}B$ as discussed in Section 3. The universal property of this limit yields

- for every non-identity morphism $f : i \to j$, an arrow $\overline{f} : M_i^{\mathcal{E}B} \to \mathcal{E}B_j$, which lets us define $M^{\mathcal{E}B}(f)$ to be $\iota_j \circ \overline{f}$; and
- an arrow $\tilde{\eta}_i : B \to M_i^{\mathcal{E}B}$ such that, for every non-identity $f : i \to j$ as in the first point, we have that $\overline{f} \circ \tilde{\eta}_i$ equals η_j.

We further define $\mathcal{E}B$ on objects by

$$\mathcal{E}B_i :\equiv \Sigma\left(m : M_i^{\mathcal{E}B}\right).\, \Sigma\left(x : B\right).\, \tilde{\eta}_i(x) = m. \tag{23}$$

This allows us to choose the canonical projection map for ι_i, and we can define $\mathcal{E}B$ on non-identity morphisms by

$$\mathcal{E}B(f) :\equiv \overline{f} \circ \iota_i. \tag{24}$$

Finally, we set

$$\eta_i(x) :\equiv (\tilde{\eta}_i(x), x, \mathsf{refl}_{\tilde{\eta}_i(x)}). \tag{25}$$

By construction, η, $\tilde{\eta}$, and ι satisfy the required naturality conditions.

▶ **Lemma 13.** *For all $i : \mathfrak{I}$, the morphism $\eta_i : B \to \mathcal{E}B_i$ is a homotopy equivalence.*

Proof. This is due to the fact that

$$\begin{aligned} \mathcal{E}B_i &\equiv \Sigma\left(m : M_i^{\mathcal{E}B}\right).\, \Sigma\left(x : B\right).\, \tilde{\eta}_i(x) = m \\ &\simeq \Sigma\left(x : B\right).\, \Sigma\left(m : M_i^{\mathcal{E}B}\right).\, \tilde{\eta}_i(x) = m \\ &\simeq B, \end{aligned} \tag{26}$$

where the last step uses that the last two Σ-components have the form of a singleton. ◀

The proceeding lemma tells us that $\mathcal{E}B$ is levelwise homotopy equivalent to the constant diagram. The crux is that, unlike the constant diagram, $\mathcal{E}B$ is Reedy fibrant by construction, i.e. a *fibrant replacement* in the usual terminology of model category theory.

▶ **Lemma 14.** *For all morphisms f in the category $\mathfrak{I}$, the fibration $\mathcal{E}B(f)$ is a homotopy equivalence.*

Proof. If $f : i \to j$ is a morphism in $\mathfrak{I}$, we have $\mathcal{E}B(f) \circ \eta_i \equiv \eta_j$ due to the naturality of η. The claim than follows by Lemma 13 as homotopy equivalences satisfy "2-out-of-3". ◀

6 The Equality Semi-simplicial Type

Let Δ_+ be the category of non-zero finite ordinals and strictly increasing maps between them. We write $[\mathsf{k}]$ for the objects, $[\mathsf{k}] \equiv \{\, 0, 1, \ldots, k \,\}$, and $[\mathsf{k}] \mathrel{+\!\!\!\rightarrow} [\mathsf{m}]$ for the hom-sets. We can now turn to our main case of interest, which is the tame category $\mathfrak{I} \equiv \Delta_+^{\mathrm{op}}$. In this case, we call $\mathcal{E}B$ the *equality semi-simplicial type* of the (given) type B. We could write down the first few values of $M_{[\mathsf{n}]}^{\mathcal{E}B}$ and $\mathcal{E}B_{[\mathsf{n}]}$ explicitly. However, these type expressions would look rather bloated. More revealing might be the homotopically equivalent presentation in Figure 2.

We think of $\mathcal{E}B_{[0]}$ as the type of points, $\mathcal{E}B_{[1]}$ as the type of lines (between two points), and of $\mathcal{E}B_{[2]}$ as the type of triangles (with its faces). The "boundary" of a triangle, as represented by $M_{[2]}$, consists of three points with three lines, and so on. In general, we think of $\mathcal{E}B_{[\mathsf{n}]}$ as (the type of) n-dimensional tetrahedra, while $M_{[\mathsf{n}]}^{\mathcal{E}B}$ are their "complete boundaries". In principle, we could have defined $\mathcal{E}B$ in a way such that Figure 2 are judgmental equalities rather than only equivalences: the stated types could be completed to form a Reedy fibrant

$$
\begin{array}{lcl}
M^{\mathcal{E}B}_{[0]} & \equiv & \mathbf{1} \\
\mathcal{E}B_{[0]} & \simeq & B \\
M^{\mathcal{E}B}_{[1]} & \simeq & B \times B \\
\mathcal{E}B_{[1]} & \simeq & \Sigma\left(b^1, b^2 : B\right).\, b^1 = b^2 \\
M^{\mathcal{E}B}_{[2]} & \simeq & \Sigma\left(b^1, b^2, b^3 : B\right).\, (b^1 = b^2) \times (b^2 = b^3) \times (b^1 = b^3) \\
\mathcal{E}B_{[2]} & \simeq & \Sigma\left(b^1, b^2, b^3 : B\right).\, \Sigma\left(p : b^1 = b^2\right).\, \Sigma\left(q : b^2 = b^3\right).\, \Sigma\left(r : b^1 = b^3\right).\, p \centerdot q = r.
\end{array}
$$

Figure 2 The "nicer" formulation of the equality semi-simplicial type. The equivalences can be shown easily using the contractibility of singletons.

diagram. However, we do not think that this is possible using a definition that is as uniform and short as the one above. Already for $\mathcal{E}B_{[3]}$, it seems unclear what the best formulation would be if we wanted to follow the presentation of Figure 2. In general, such a construction would most likely make use of Street's orientals [25].

For any $[\mathsf{n}]$, the co-slice category $[\mathsf{n}]/\Delta_+^{\mathrm{op}}$ is a poset. This is a consequence of the fact that all morphisms in Δ_+ are monic. We have the forgetful functor $U : [\mathsf{n}]/\Delta_+^{\mathrm{op}} \to \Delta_+^{\mathrm{op}}$. Further, $[\mathsf{n}]/\Delta_+^{\mathrm{op}}$ is isomorphic to the poset $\mathcal{P}_+([\mathsf{n}])$ of nonempty subsets of the set $[\mathsf{n}] \equiv \{0, 1, \ldots, n\}$, where we have an arrow between two subsets if the first is a superset of the second. The downwards closed full subcategories of $[\mathsf{n}]/\Delta_+^{\mathrm{op}}$ correspond to downwards closed subsets of $\mathcal{P}_+([\mathsf{n}])$. If S is such a downwards closed subset, we write $\mathsf{lim}_S(\mathcal{E}B \circ U)$, omitting the implied functor $S \to [\mathsf{n}]/\Delta_+^{\mathrm{op}}$.

Any set $s \subseteq [\mathsf{n}]$ generates such a downwards closed set for which we write $\overline{s} :\equiv \mathcal{P}_+(s)$. For $k \in s$, we write $\overline{s}_{-k}$ for the set that we get if we remove exactly two sets from $\overline{s}$, namely s itself and the set $s - k$ (i.e. s without the element k). We call $\mathsf{lim}_{\overline{[\mathsf{n}]}_{-k}}(\mathcal{E}B \circ U)$ the *k-th n-horn*.

▶ **Main Lemma 15.** *For any $n \geq 1$ and $k \in [\mathsf{n}]$, call the fibration from the full n-dimensional tetrahedron to the k-th n-horn*

$$\mathsf{lim}_{\overline{[\mathsf{n}]}}(\mathcal{E}B \circ U) \twoheadrightarrow \mathsf{lim}_{\overline{[\mathsf{n}]}_{-k}}(\mathcal{E}B \circ U) \tag{27}$$

a horn-filler fibration. *All horn-filler fibrations are homotopy equivalences.*

▶ **Remark and Corollary 16** (Types are Kan complexes)**.** As both Steve Awodey and an anonymous reviewer of have pointed out to me, Main Lemma 15 can be seen as a simplicial variant of Lumsdaine's [16] and van den Berg-Garner's [28] result that types are weak ω-groupoids. Both of these (independent) articles use Batanin's [5] definition, slightly modified by Leinster [15], of a weak ω-groupoid.

Let us make the construction of a simplicial weak ω-groupoid, i.e. of a Kan complex, concrete. We can do this for the assumed type-theoretic fibration category $\mathfrak{C}$ as long as it is locally small (i.e. all hom-sets are sets). As before, we can without loss of generality assume that the type we want to consider lives in the empty context, i.e. is given by an object B. We can define a semi-simplicial set

$$S : \Delta_+^{\mathrm{op}} \to \mathsf{Set} \tag{28}$$

$$S_{[\mathsf{n}]} :\equiv \mathfrak{C}(\mathbf{1}, \mathcal{E}B_{[\mathsf{n}]}). \tag{29}$$

For a morphism f of Δ_+^{op}, the functor S is given by simply composing with $\mathcal{E}B(f)$.

Shulman's *acyclic fibration lemma* [24, Lemma 3.11], applied on the result of our Main Lemma 15, gives us sections of all horn-filler fibrations. Therefore, S satisfies the Kan condition. By a result Rourke and Sanderson [21] (see also McClure [19] for a combinatorical proof), such a semi-simplicial set can be given the structure of a Kan *simplicial* set, an incarnation of a weak ω-groupoid.

To get the result that types in HoTT are Kan complexes, we simply take $\mathfrak{C}$ to be the syntactic category of HoTT, where we have to assume strict η for Π, Σ and $\mathbf{1}$. This allows us to say very concretely that the terms of the types that we can write down form a Kan complex.

Proof of Main Lemma 15. Fix $[\mathsf{n}]$. We show more generally that, for any $s \subseteq [\mathsf{n}]$ with cardinality $|s| \geq 2$ and $k \in s$, the fibration

$$\mathsf{lim}_{\overline{s}}(\mathcal{E}B \circ U) \twoheadrightarrow \mathsf{lim}_{\overline{s}_{-k}}(\mathcal{E}B \circ U) \tag{30}$$

is an equivalence. Note that $\mathsf{lim}_{\overline{s}}(\mathcal{E}B \circ U)$ is isomorphic to $\mathcal{E}B_{[|\mathsf{s}|-1]}$.

The proof is performed by induction on the cardinality of s. If s has only one element apart from k, then $\overline{s}_{-k}$ is the one-object category $\{\{k\}\}$ and we have

$$\mathsf{lim}_{\{\{k\}\}}(\mathcal{E}B \circ U) \cong \mathcal{E}B_{[0]}. \tag{31}$$

The statement then follows from Lemma 14.

Let us explain the induction step. The inclusions $\{\{k\}\} \subseteq \overline{s}_{-k} \subset \overline{s}$ give rise to a triangle

$$\begin{array}{ccc} \mathsf{lim}_{\overline{s}}(\mathcal{E}B \circ U) & \twoheadrightarrow & \mathsf{lim}_{\overline{s}_{-k}}(\mathcal{E}B \circ U) \\ & \searrow & \downarrow \\ & & \mathsf{lim}_{\{\{k\}\}}(\mathcal{E}B \circ U) \end{array}$$

of fibrations. The top horizontal fibration is the one of which we want to prove that it is an equivalence. Using "2-out-of-3" and the fact that the left (diagonal) fibration is an equivalence by Lemma 14, it is sufficient to show that the right vertical fibration is an equivalence. To do this, we decompose it into $2^{|s|-1} - 1$ fibrations, each of which can be viewed as the pullback of a smaller horn-filler fibration:

Consider the set $\mathcal{P}_+(s - k)$ of those nonempty subsets of s that do not contain k. The number of those is $2^{|s|-1} - 1$. We label those sets as $\alpha_1, \alpha_2, \ldots, \alpha_{2^{|s|-1}-1}$, where the order is arbitrary with the only condition that their cardinality is nondecreasing, i.e. $i < j$ implies $|\alpha_i| < |\alpha_j|$.

We further define $2^{|s|-1}$ subsets of $\mathcal{P}_+(s)$, named $S_0, S_1, \ldots, S_{2^{|s|-1}}$. Define S_0 to be $\{\{k\}\}$. Then, define S_i to be S_{i-1} with two additional elements, namely α_i and $\alpha_i \cup \{k\}$. In this process, every element of $\mathcal{P}_+(s)$ is clearly added exactly once. In particular, $S_{2^{|s|-1}} \equiv \overline{s}$ and $S_{2^{|s|-1}-1} \equiv \overline{s}_{-k}$. Further, all S_i are downwards closed, which is easily seen to be the case by induction on i: it is the case for $i \equiv 0$, and in general, S_i contains all proper subsets of $\alpha_i \cup \{k\}$ due to the single ordering condition that we have put on the sequence (α_j).

It is easy to see that

$$S_i \;\equiv\; S_{i-1} \cup \overline{\alpha_i \cup \{k\}} \tag{32}$$

$$\overline{\alpha_i \cup \{k\}}_{-k} \;\equiv\; S_{i-1} \cap \overline{\alpha_i \cup \{k\}}. \tag{33}$$

By Lemma 11, we thus have a pullback square

$$\begin{array}{ccc} \mathsf{lim}_{S_i}(\mathcal{E}B \circ U) & \twoheadrightarrow & \mathsf{lim}_{\overline{\alpha_i \cup \{k\}}}(\mathcal{E}B \circ U) \\ \downarrow & & \downarrow \\ \mathsf{lim}_{S_{i-1}}(\mathcal{E}B \circ U) & \twoheadrightarrow & \mathsf{lim}_{\overline{\alpha_i \cup \{k\}}_{-k}}(\mathcal{E}B \circ U) \end{array}$$

For $i \leq 2^{|s|-1} - 2$, the right vertical morphism is a homotopy equivalence by the induction hypothesis. As acyclic fibrations are stable under pullback, the left vertical morphism is one as well. As the composition of equivalences is an equivalence, we conclude that

$$\mathsf{lim}_{\overline{s}_{-k}}(\mathcal{E}B \circ U) \twoheadrightarrow \mathsf{lim}_{\{\{k\}\}}(\mathcal{E}B \circ U) \tag{34}$$

is indeed an equivalence. ◀

▶ Remark. Recall that a simplicial object $X : \Delta^{\mathrm{op}} \to \mathcal{D}$ satisfies the *Segal condition* (see [22]) if the "fibration"

$$X_{[\mathsf{n}]} \to \underbrace{X_{[1]} \times_{X_{[0]}} X_{[1]} \times_{X_{[0]}} \dots \times_{X_{[0]}} X_{[1]}}_{n \text{ factors}} \tag{35}$$

is an equivalence. In our situation, it looks as if it was easy to check the Segal condition; more precisely, a shorter argument than the one in the proof could show that *all* the fibrations of the form (35)) are homotopy equivalences. Our construction with the sequence $\alpha_1, \alpha_2, \dots, \alpha_{2^{|s|-1}-1}$ seems to contain a "manual" proof of the fact that checking this form of the Segal condition would be sufficient.

7 Fibrant Diagrams of Natural Transformations

Let us first formalise what we mean by the "type of natural transformations between two diagrams". If I is a tame category and $D, E : I \to \mathfrak{C}$ are Reedy fibrant diagrams, the exponential $E^D : I \to \mathfrak{C}$ in the functor category $\mathfrak{C}^I$ exists and is Reedy fibrant [24, Theorem 11.11] and thus has a limit in $\mathfrak{C}$. What we are interested in is the more general case that D might not be fibrant, but we also do not need any exponential.[1] On a more abstract level, what we want to do can be described as follows. For any downwards closed subcategory of I, we consider the exponential of D and E restricted to this subcategory, and take its limit. We basically construct approximations to the "type of natural transformations" from D to E which, in fact, corresponds to the limit of these approximations, should it exist. Fortunately, it is easy to do everything "by hand" on a very basic level.

We write $(\!|I|\!)$ for the underlying partially ordered set of I that we get if we make any two parallel arrows equal (we "truncated" all hom-sets). This makes sense even if I is not inverse, but if it is, then so is $(\!|I|\!)$. There is a canonical functor $|-|_I : I \to (\!|I|\!)$. As the objects of I are the same as those of $(\!|I|\!)$, we omit this functor when applied to an object, i.e. for $i \in I$ we write $i \in (\!|I|\!)$ instead of $|i|_I \in (\!|I|\!)$.

[1] The author expects that the exponential E^D exists and is fibrant even if only E is fibrant (note that D is automatically at least *pointwise* fibrant, as all objects in $\mathfrak{C}$ are fibrant by definition). This would lead to an alternative representation of the same construction, but the author has decided to use the less abstract one presented here as it seems to give a more direct argument.

▶ **Definition 17** (Diagram of Natural Transformations)**.** Given an inverse category I, a diagram $D : I \to \mathfrak{C}$ and a fibrant diagram $E : I \to \mathfrak{C}$ with

$$E_i \equiv \Sigma\left(m : M_i^E\right).\, F^E_{(i,m)} \tag{36}$$

as introduced in Definition 6, we define a fibrant diagram $N : (\!|I|\!) \to \mathfrak{C}$ together with a natural transformation

$$v : \left((N \circ |-|_I) \times D\right) \to E \tag{37}$$

simultaneously, where $(N \circ |-|_I) \times D$ is the functor $I \to \mathfrak{C}$ that is given by taking the product pointwise.

Assume i is an object in I. Assume further that we have defined both N and v for all predecessors of i (i.e. N is defined on $\{\, x \in (\!|I|\!) \mid x \prec i \,\}$ and v is defined on $\{\, x \in I \mid x \prec i \,\}$). v then gives rise to a map

$$\overline{v} : \mathsf{lim}_{\{\, x \in (\!|I|\!) \mid x \prec i \,\}} N \times D_i \to M_i^E. \tag{38}$$

Using the very simple Lemma 9, we have $\mathsf{lim}_{\{\, x \in (\!|I|\!) \mid x \prec i \,\}} N \cong \mathsf{lim}_{i /\!\!/ (\!|I|\!)}(N \circ U) \cong M_i^N$. We define $N_i \equiv \Sigma\left(m : M_i^N\right).\, F^N_{(i,m)}$ by choosing the fibre over m to be

$$F^N_{(i,m)} :\equiv \Pi_{d : D_i} F^E(i, \overline{v}(m, d)). \tag{39}$$

This definition also gives a canonical morphism $v_i : N_i \times D_i \to E_i$ which extends v.

Let us apply this construction to define the type of constant functions between types A and B in the way that we already suggested in Figure 1 on page 118. First, we define the $[0]$-*coskeleton* of the diagram that is constantly A, which we also have referred to as the *trivial diagram* over A, as the functor $\mathcal{T}A : \Delta_+^{\mathrm{op}} \to \mathfrak{C}$ as follows. For objects, it is simply given by

$$\mathcal{T}A_{[\mathsf{k}]} :\equiv \underbrace{A \times A \times \ldots \times A}_{(k+1)\ \mathit{factors}}. \tag{40}$$

If we view an element of $\mathcal{T}A_{[\mathsf{i}]}$ as a function $[\mathsf{i}] \to A$, for a map $f : [\mathsf{i}] \xrightarrow{+} [\mathsf{j}]$ we get $\mathcal{T}A(f) : \mathcal{T}A_{[\mathsf{j}]} \to \mathcal{T}A_{[\mathsf{i}]}$ by composition with f. We then define the functor $\mathcal{N}_{A,B} : (\!|\Delta_+^{\mathrm{op}}|\!) \to \mathfrak{C}$ via the above construction as the "fibrant diagram of natural transformations" from $\mathcal{T}A$ to $\mathcal{E}B$. Note that $(\!|\Delta_+^{\mathrm{op}}|\!)$ is isomorphic to ω^{op}. Using the homotopy equivalent formulation of $\mathcal{E}B$ stated in (2) and the definitions of const and coh of Section 2, we get

$$\mathcal{N}_{A,B}([0]) \simeq (A \to B) \tag{41}$$

as well as

$$\mathcal{N}_{A,B}([1]) \simeq \Sigma\,(f : A \to B).\, \mathsf{const}_f \tag{42}$$

and

$$\mathcal{N}_{A,B}([2]) \simeq \Sigma\,(f : A \to B).\, \Sigma\,(c : \mathsf{const}_f).\, \mathsf{coh}_{f,c}. \tag{43}$$

We want to stress the intuition that we think of functions with an infinite tower of coherence condition by introducing the following notation:

▶ **Definition 18** ($A \xrightarrow{\omega} B$)**.** Given types A and B, we write $A \xrightarrow{\omega} B$ synonymously for $\mathsf{lim}_{(\!|\Delta_+^{\mathrm{op}}|\!)} \mathcal{N}_{A,B}$.

We usually omit the indices of $\mathcal{N}_{A,B}$ and just write $\mathcal{N}$, provided that A, B are clear from the context. This allows us write $\mathcal{N}_{[\mathsf{n}]}$ instead of $\mathcal{N}_{A,B}([\mathsf{n}])$.

Analogously to Definition 18, let us write the following:

▶ **Definition 19** ($A \xrightarrow{[\mathsf{n}]} B$)**.** Given types A and B and a (usually finite) number n, we write $A \xrightarrow{[\mathsf{n}]} B$ synonymously for $\mathcal{N}_{[\mathsf{n}]}$. To enable a uniform presentation, we define $A \xrightarrow{[-1]} B$ to be the unit type.

We are now able to make the main goal, as outlined in Section 2.3, precise: we will construct a function $(\|A\| \to B) \to (A \xrightarrow{\omega} B)$ and prove that it is a homotopy equivalence. For now, let us record that we can get a function $B \to (A \xrightarrow{\omega} B)$. In the following definition, we use the cones $\eta : B \to \mathcal{E}B$ and $\tilde{\eta} : B \to M^{\mathcal{E}B}$ from Section 5.

▶ **Definition 20** (Canonical function $s : B \to (A \xrightarrow{\omega} B)$)**.** Define a cone $\gamma : B \to \mathcal{N}$ which maps $b : B$ to the function that is "judgmentally constantly b", in the following way. First, notice that the matching object $M^{\mathcal{N}}_{[\mathsf{n}]}$ is simply $\mathcal{N}_{[\mathsf{n}-1]}$ (due to the fact that $(\!|\Delta^{\mathrm{op}}_+|\!)$ is a total order). Assume we have already defined the component $\gamma_{[\mathsf{n}-1]} : B \to \mathcal{N}_{[\mathsf{n}-1]}$ such that $\overline{v}(\gamma_{[\mathsf{n}-1]}(b), x) \equiv \tilde{\eta}_{[\mathsf{n}]}(b)$, with $\overline{v}$ as in (38), for all $x : \mathcal{T}A_{[\mathsf{n}]}$. We can then define $\gamma_{[\mathsf{n}]}(b)$ by giving an element of $F^{\mathcal{N}}([\mathsf{n}], \gamma_{[\mathsf{n}-1]}(b))$, but that expression evaluates to $\Pi_{x:\mathcal{T}A_{[\mathsf{n}]}} \Sigma\,(x : B)\,.\,\tilde{\eta}_{[\mathsf{n}]}(x) = \tilde{\eta}_{[\mathsf{n}]}(b)$. Thus, we can take $\gamma_{[\mathsf{n}]}(b)$ to be

$$\gamma_{[\mathsf{n}]}(b) :\equiv \big(\gamma_{[\mathsf{n}-1]}(b), \lambda z.(b, \mathsf{refl}_{\tilde{\eta}_{[\mathsf{n}-1]}(b)})\big). \tag{44}$$

It is straightforward to check that the condition $\overline{v}(\gamma_{[\mathsf{n}]}, x) \equiv \tilde{\eta}_{[\mathsf{n}+1]}(b)$ is preserved. Define the function $s : B \to (A \xrightarrow{\omega} B)$ to be $\mathsf{lim}_{(\!|\Delta^{\mathrm{op}}_+|\!)}\gamma$, the arrow that is induced by the universal property of the limit.

8 Extending Semi-Simplicial Types

In this section, we first define the category $\widehat{\Delta}_+$. We can then view $\widehat{\Delta}^{\mathrm{op}}_+$ as an extension of Δ^{op}_+, as Δ^{op}_+ can be embedded into $\widehat{\Delta}^{\mathrm{op}}_+$, and this embedding has a retraction R with the property that the co-slice $c/\widehat{\Delta}^{\mathrm{op}}_+$ is always isomorphic to $R_c/\Delta^{\mathrm{op}}_+$. With the help of this category, we can describe precisely how we want to apply our "expanding and contracting" strategy. The definition of $\widehat{\Delta}_+$ is motivated by the proofs of Propositions 2 and 3, and this will become clear when we show how exactly we use $\widehat{\Delta}_+$, see especially Figure 4.

▶ **Definition 21** ($\widehat{\Delta}_+$)**.** Let $\widehat{\Delta}_+$ be the following category. For every object $[\mathsf{k}]$ of Δ_+ (i.e. every natural number k), and every number $i \in [\mathsf{k}+1]$, we have an object $c^i_{[\mathsf{k}]}$. Given objects $c^i_{[\mathsf{k}]}$ and $c^j_{[\mathsf{m}]}$, we define $\widehat{\Delta}_+\left(c^i_{[\mathsf{k}]}, c^j_{[\mathsf{m}]}\right)$ to a subset of the set of maps $\Delta_+([\mathsf{k}], [\mathsf{m}])$. It is given by

$$\widehat{\Delta}_+\left(c^i_{[\mathsf{k}]}, c^j_{[\mathsf{m}]}\right) :\equiv \left\{\, f : [\mathsf{k}] \xrightarrow{+} [\mathsf{m}] \;\middle|\; \alpha(k, m, i, j) \,\right\} \tag{45}$$

where the condition α is defined as

$$\alpha(k, m, i, j) :\equiv \begin{cases} f(x) \equiv x \text{ for all } x < i, \text{ and } f(x) > x \text{ for all } x \geq i & \text{if } i < j \\ f(x) \equiv x \text{ for all } x < i & \text{if } i \equiv j \\ \bot & \text{if } i > j. \end{cases} \tag{46}$$

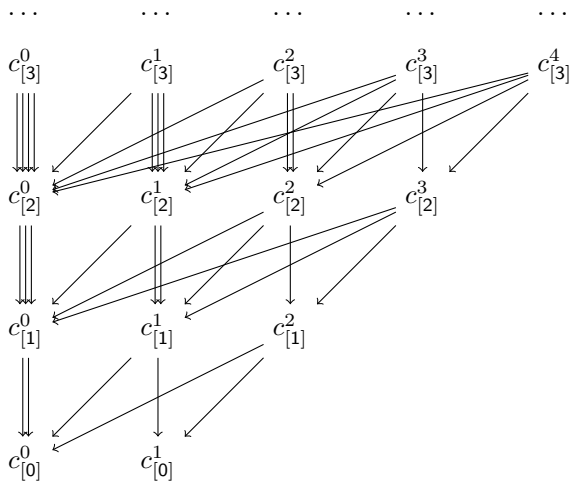


Figure 3 The category $\widehat{\Delta}_+^{\mathrm{op}}$.

What will be useful for us is the opposite category $\widehat{\Delta}_+^{\mathrm{op}}$. A part of it, namely the subcategory $\left\{ c^i_{[\mathsf{k}]} \in \widehat{\Delta}_+^{\mathrm{op}} \;\middle|\; k \leq 3 \right\}$, can be pictured as shown in Figure 3. We only draw the "generating" arrows $c^j_{[\mathsf{m}+1]} \to c^i_{[\mathsf{m}]}$.

The idea is that the full subcategory of objects $c^0_{[\mathsf{m}]}$ is exactly Δ_+, and that every object $c^i_{[\mathsf{m}]}$ in $\widehat{\Delta}_+$ receives exactly one arrow for every $[\mathsf{k}] \overset{+}{\to} [\mathsf{m}]$. We make this precise as follows:

▶ **Lemma 22.** *The canonical embedding $\Delta_+^{\mathrm{op}} \hookrightarrow \widehat{\Delta}_+^{\mathrm{op}}$, defined by $[\mathsf{m}] \mapsto c^0_{[\mathsf{m}]}$, has a retraction*

$$R : \widehat{\Delta}_+^{\mathrm{op}} \to \Delta_+^{\mathrm{op}} \tag{47}$$

$$R(c^j_{[\mathsf{m}]}) :\equiv [\mathsf{m}] \tag{48}$$

and, for all objects $c^j_{[\mathsf{m}]}$ in $\widehat{\Delta}_+^{\mathrm{op}}$, the functor that R induces on the co-slice categories

$$c^j_{[\mathsf{m}]}/\widehat{\Delta}_+^{\mathrm{op}} \;\to\; [\mathsf{m}]/\Delta_+^{\mathrm{op}} \tag{49}$$

is an isomorphism of categories.

Proof. It is clear that $\Delta_+^{\mathrm{op}} \hookrightarrow \widehat{\Delta}_+^{\mathrm{op}} \xrightarrow{R} \Delta_+^{\mathrm{op}}$ is the identity on Δ_+^{op}. For any $c^j_{[\mathsf{m}]}$, fix an object $[\mathsf{k}]$ in Δ_+^{op} and take a morphism $f : [\mathsf{k}] \overset{+}{\to} [\mathsf{m}]$. There is exactly one i such that the condition $\alpha(k, m, i, j)$ in (46) is fulfilled. This proves the second claim. ◀

Let us extend the functor $\mathcal{T}A : \Delta_+^{\mathrm{op}} \to \mathfrak{C}$ (see Section 7) to the whole category $\widehat{\Delta}_+^{\mathrm{op}}$. Assume that a type A is given. We want to define a diagram $\widehat{\mathcal{T}}A$ that extends $\mathcal{T}A$. This corresponds to the point where, in Section 2, we had assumed that a point $\mathfrak{a}_{\mathsf{o}} : A$ was given, in other words, we had added $(\mathfrak{a}_{\mathsf{o}} : A)$ to the context. We do the same here. Recall that we write $(\mathfrak{C}/A)_{\mathsf{f}}$ for the type-theoretic fibration category with fixed context A, as explained in Section 3. The diagram that we define is a functor

$$\widehat{\mathcal{T}}A : \widehat{\Delta}_+^{\mathrm{op}} \to (\mathfrak{C}/A)_{\mathsf{f}}\,. \tag{50}$$

In order to be closer to the type-theoretic notation and to hopefully increase readability, we write objects of $(\mathfrak{C}/A)_{\mathsf{f}}$ simply as $B(\mathfrak{a}_{\mathsf{o}})$ if they are of the form $\Sigma\,(a : A)\,.\,B(a) \twoheadrightarrow A$. This

uses that we can do the whole construction fibrewise, i.e. that we can indeed assume a fixed but arbitrary $\mathfrak{a}_\circ : A$ "in the context". Of course, objects in $(\mathfrak{C}/A)_\mathsf{f}$ of the form $A \times B \twoheadrightarrow A$ are simply denoted by B.

Using this notation, we define $\widehat{\mathcal{T}}A$ on objects by

$$\widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]}) :\equiv \underbrace{A \times A \times \ldots \times A}_{(m+1-j)\text{ factors}}, \tag{51}$$

for which we simply write A^{m+1-j}. Given $c^j_{[\mathsf{m}]} \xrightarrow{f} c^i_{[\mathsf{k}]}$ in $\widehat{\Delta}^{\mathrm{op}}_+$, we thus need to define a map $\widehat{\mathcal{T}}A(f) : A^{m+1-j} \to A^{k+1-i}$. As in the definition of $\mathcal{T}A$, the map $f : [\mathsf{k}] \xrightarrow{+} [\mathsf{m}]$ gives rise to a function $\overline{f} : A^{m+1} \to A^{k+1}$ by "composition". We define $\widehat{\mathcal{T}}A(f)$ as the composite

$$\begin{array}{c} A^{m+1-j} \\ \Big\downarrow \vec{a} \mapsto (\underbrace{\mathfrak{a}_\circ, \mathfrak{a}_\circ, \ldots, \mathfrak{a}_\circ}_{j \text{ times } \mathfrak{a}_\circ}, \vec{a}) \\ A^j \times A^{m+1-j} \\ \Big\downarrow \overline{f} \\ A^i \times A^{k+1-i} \\ \Big\downarrow \mathsf{snd} \\ A^{k+1-i} \end{array}$$

We have a diagram $\mathcal{E}B \circ R : \widehat{\Delta}^{\mathrm{op}}_+ \to \mathfrak{C}$, which we can (pointwise) pull back along $A \twoheadrightarrow \mathbf{1}$, which gives us a diagram that we call $\widehat{\mathcal{E}}B : \widehat{\Delta}^{\mathrm{op}}_+ \to (\mathfrak{C}/A)_\mathsf{f}$. This diagram is Reedy fibrant. With the construction of Section 7, we can define $\widehat{\mathcal{N}} : (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) \to (\mathfrak{C}/A)_\mathsf{f}$ to be the "fibrant diagram of natural transformations" from $\widehat{\mathcal{T}}A$ to $\widehat{\mathcal{E}}B$.

We can picture $\widehat{\mathcal{N}}$ on the subcategory $\left\{ c^j_{[\mathsf{m}]} \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) \;\middle|\; m \leq 2 \right\}$ as shown in Figure 4. For readability, we use the homotopy equivalent representation of the values of $\mathcal{E}B$ as shown in Figure 2. Further, we only write down the values of $F^{\widehat{\mathcal{E}}B}$ (i.e. the fibres) instead of the full expression $\widehat{\mathcal{E}}B(c^j_{[\mathsf{m}]}) \equiv \Sigma\left(t : M^{\widehat{\mathcal{E}}B}(c^j_{[\mathsf{m}]})\right).F^{\widehat{\mathcal{E}}B}(c^j_{[\mathsf{m}]}, t)$. For example, $\mathsf{const}_f \twoheadrightarrow (f : A \to B)$ stands for the projection $\Sigma\,(f : A \to B).\,\mathsf{const}_f \twoheadrightarrow (A \to B)$. The reader is invited to make a comparison with Proposition 3. Recall that, in the proof of Proposition 3, we have started with the Σ-component f_1. In the "expanding" part, we have added the pair of f and c_1, which (together) form a contractible type, as well as the pair of c and d_1, and c_2 and d_3. We have also used that the types of d and d_2 are, in the presence of the other Σ-components, contractible. Then, in the "retracting" part, we have used that the types of d_3 and d_1 are contractible, and that c_1 and d_2, as well as f_1 and c_2, form pairs of two other contractible types.

To compare $\widehat{\mathcal{N}}$ with $\mathcal{N}$, first note that $\mathcal{N} : \Delta^{\mathrm{op}}_+ \to \mathfrak{C}$ can be pulled back along $A \twoheadrightarrow \mathbf{1}$ pointwise and yields a diagram $\Delta^{\mathrm{op}}_+ \to (\mathfrak{C}/A)_\mathsf{f}$. This diagram is identical (pointwise isomorphic) to the diagram that we get if we first pull back the diagrams $\mathcal{T}A$ and $\mathcal{E}B$, and then take the "fibrant diagram of natural transformations". Further, as "limits commute with limits", the limit of this diagram is, in $(\mathfrak{C}/A)_\mathsf{f}$, isomorphic to the pullback of $A \xrightarrow{\omega} B$ along $A \twoheadrightarrow \mathbf{1}$. It is thus irrelevant at which point in the construction we "add $(\mathfrak{a}_\circ : A)$ to the context", i.e. at which point we switch from $\mathfrak{C}$ to the slice over A. This allows us to compare constructions in $(\mathfrak{C}/A)_\mathsf{f}$ and $\mathfrak{C}$, by implicitly pulling back the latter. As it is easy to see, $\widehat{\mathcal{N}}$ extends $\mathcal{N}$ in this sense (i.e. $\widehat{\mathcal{N}}(c^0_{[\mathsf{m}]})$ is the pullback of $\mathcal{N}_{[\mathsf{m}]}$ along $A \twoheadrightarrow \mathbf{1}$).

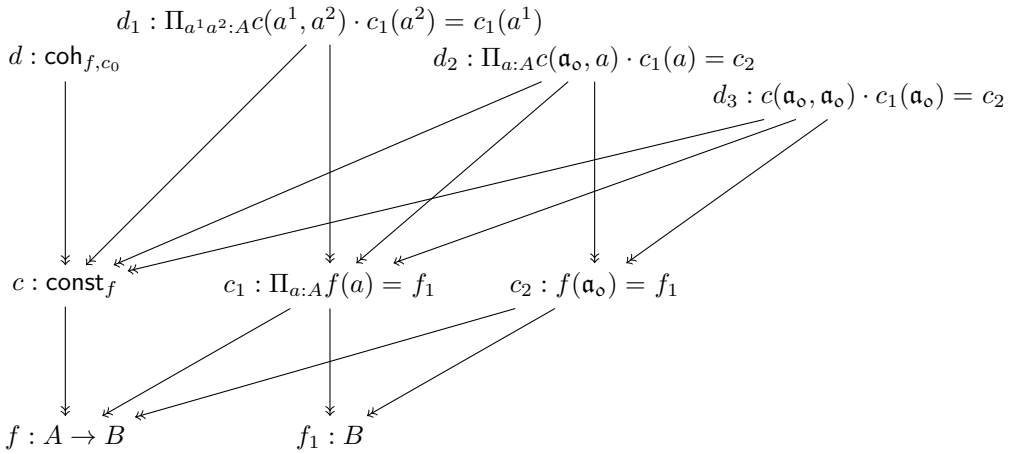


Figure 4 The diagram $\widehat{\mathcal{N}}$ in readable (homotopy equivalent) representation; only the three lowest levels (the images of $c^j_{[\mathsf{m}]}$ with $m \leq 2$) are drawn. Note that we use that same identifiers as in the proofs of Propositions 2 and 3.

Recall that we have defined a cone $\gamma : B \to \mathcal{N}$ and an arrow $s : B \to (A \xrightarrow{\omega} B)$ in Definition 20. Exploiting that $\gamma_{[\mathsf{n}]}(b)$ was defined in a way that makes it completely independent of the "argument" $x : \mathcal{T}A_{[\mathsf{n}]}$, and using Lemma 22, we can extend γ to a cone $\overline{\gamma} : B \to \widehat{\mathcal{N}}$, essentially by putting $\overline{\gamma}_{c^j_{[\mathsf{m}]}} :\equiv \gamma_{[\mathsf{m}]}$. This gives a morphism

$$\overline{s} : B \to \mathsf{lim}_{(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)} \widehat{\mathcal{N}} \tag{52}$$

which extends s, in the sense that (the pullback of) s is the composition

$$B \xrightarrow{\overline{s}} \mathsf{lim}_{(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)} \widehat{\mathcal{N}} \xrightarrow{\mathsf{pr}} \mathsf{lim}_{(\!|\Delta^{\mathrm{op}}_+|\!)} \mathcal{N}, \tag{53}$$

with pr coming from the embedding $(\!|\Delta^{\mathrm{op}}_+|\!) \hookrightarrow (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)$ and the fact that the restriction of $\widehat{\mathcal{N}}$ to $\{c^0_{[\mathsf{m}]}\}$ is $\mathcal{N}$ (pulled back along $A \twoheadrightarrow \mathbf{1}$; note that the codomain of pr is implicitly pulled back as well). Further, observing that $\widehat{\mathcal{N}}(c^1_{[\mathsf{0}]})$ is canonically equivalent to B (as used in Figure 4), the composition

$$B \xrightarrow{\overline{s}} \mathsf{lim}_{(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)} \widehat{\mathcal{N}} \xrightarrow{\mathsf{pr}'} \widehat{\mathcal{N}}(c^1_{[\mathsf{0}]}) \xrightarrow{\sim} B \tag{54}$$

is the identity on B.

9 The Main Theorem

The preparations of the previous sections allow us prove our main result. We proceed analogously to our arguments for the special cases in Section 2: Lemma 23 and Corollary 24 show that certain fibrations are homotopy equivalences, i.e. that certain types are contractible. This is then used in Main Lemma 25 to perform the "expanding and contracting" argument, which shows that, if we assume a point in A, the function s from Definition 20 is a homotopy equivalence. Admittedly, especially Lemma 23 requires extensive calculations.

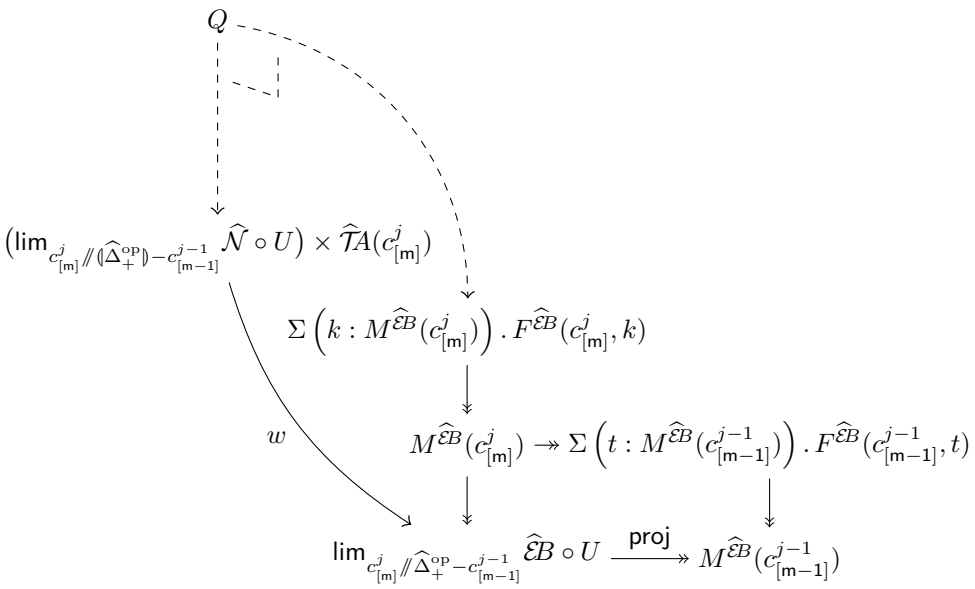


Figure 5 Derivation of a homotopy equivalence

We need to keep the extremely simple statement of Lemma 9 in mind: the limit of $\widehat{\mathcal{N}} \circ U$ restricted to $z /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|)$ is isomorphic to the limit of $\widehat{\mathcal{N}}$ restricted to $\left\{ y \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|) \;\middle|\; y \prec z \right\}$. We prefer the slightly more concise first notation.

For the following statement, note that $\widehat{\mathcal{N}}(c^j_{[\mathsf{m}]})$ is the same as $\mathsf{lim}_{c^j_{[\mathsf{m}]} / (\!|\widehat{\Delta}_+|)} \widehat{\mathcal{N}} \circ U$.

▶ **Lemma 23.** *The fibration*

$$\widehat{\mathcal{N}}(c^j_{[\mathsf{m}]}) \twoheadrightarrow \mathsf{lim}_{\left\{ x \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|) \;\middle|\; x \prec c^j_{[\mathsf{m}]}, x \neq c^{j-1}_{[\mathsf{m}-1]} \right\}} \widehat{\mathcal{N}} \tag{55}$$

is a homotopy equivalence for any $[\mathsf{m}]$ *and* j*.*

Proof. There is a single morphism in $\widehat{\Delta}^{\mathrm{op}}_+ \left(c^j_{[\mathsf{m}]}, c^{j-1}_{[\mathsf{m}-1]}\right)$. For the category $c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+$ where this morphism is removed, we write $c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}$. The fibration (55) can then be written as

$$\widehat{\mathcal{N}}(c^j_{[\mathsf{m}]}) \twoheadrightarrow \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}_+|) - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{N}} \circ U. \tag{56}$$

By construction of $\widehat{\mathcal{N}}$, we have a natural transformation $v : (\widehat{\mathcal{N}} \circ |-|_{\widehat{\Delta}^{\mathrm{op}}_+}) \times \widehat{\mathcal{T}}A \to \widehat{\mathcal{E}}B$, which gives rise to a morphism

$$w : \left(\mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|) - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{N}} \circ U\right) \times \widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]}) \;\to\; \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{E}}B \circ U. \tag{57}$$

Consider the diagram shown in Figure 5, in which Q is defined to be the pullback.

The right part (everything without the leftmost column) of that diagram comes from applying the functor $\mathsf{lim}_{-}(\widehat{\mathcal{E}}B \circ U)$ to the diagram in $\mathsf{Sub}(c^j_{[\mathsf{m}]} / \widehat{\Delta}^{\mathrm{op}}_+)$ that is shown in Figure 6.

In Figure 5, the fibration labelled proj comes of course from

$$\left(c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}\right) \supset \left(c^{j-1}_{[\mathsf{m}-1]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+\right), \tag{58}$$

$$\begin{array}{ccc} c^j_{[\mathsf{m}]}/\widehat{\Delta}^{\mathrm{op}}_+ & & \\ \downarrow & & \\ c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ & \longrightarrow & c^{j-1}_{[\mathsf{m}-1]}/\widehat{\Delta}^{\mathrm{op}}_+ \\ \downarrow & & \downarrow \\ c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]} & \longrightarrow & c^{j-1}_{[\mathsf{m}-1]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ \end{array}$$

Figure 6 A small diagram in $\mathsf{Sub}(c^j_{[\mathsf{m}]}/\widehat{\Delta}^{\mathrm{op}}_+)$. This uses the principle that, in an inverse category $\mathfrak{I}$ with a morphism $k : x \to y$, the categories $k /\!\!/ (x /\!\!/ \mathfrak{I})$ and $y /\!\!/ \mathfrak{I}$ are isomorphic.

as shown in Figure 6. We give it a name solely to make referencing it easier. Our goal is to derive a representation of Q. As the right square is a pullback square by Lemma 11, we have

$$M^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}) \;\cong\; \Sigma\big(t : \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{E}B}\big).\, F^{\widehat{\mathcal{E}B}}(c^{j-1}_{[\mathsf{m}-1]}, \mathsf{proj}(t)). \tag{59}$$

Using this, can write the top expression of the middle column as

$$\begin{aligned} & \Sigma\big(k : M^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]})\big).\, F^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}, k) \\ \simeq\; & \Sigma\big(t : \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{E}B}\big).\, \Sigma\big(n : F^{\widehat{\mathcal{E}B}}(c^{j-1}_{[\mathsf{m}-1]}, \mathsf{proj}(t))\big).\, F^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}, (t, n)). \end{aligned} \tag{60}$$

The pullback Q is thus

$$\begin{aligned} \Sigma\big(p : \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{N}} \circ U\big).\, \Sigma\big(a : \widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]})\big). & \\ \Sigma\big(n : F^{\widehat{\mathcal{E}B}}(c^{j-1}_{[\mathsf{m}-1]}, \mathsf{proj}(w(p,a)))\big). & \\ F^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}, (w(p,a), n)). & \end{aligned} \tag{61}$$

The composition of the two vertical fibrations in the middle column is a homotopy equivalence by Main Lemma 15 and Lemma 22. As acyclic fibrations are stable under pullback, the fibration

$$Q \;\twoheadrightarrow\; \big(\mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{N}}\big) \times \widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]}) \tag{62}$$

is a homotopy equivalence as well. Function extensionality implies that a family of contractible types is contractible (i.e. that acyclic fibrations are preserved by Π), and we get that the first projection

$$\begin{array}{l} \Sigma\big(p : \mathsf{lim}_{\left\{\, x \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) \;\middle|\; x \prec c^j_{[\mathsf{m}]}, x \not\equiv c^{j-1}_{[\mathsf{m}-1]} \,\right\}} \widehat{\mathcal{N}}\big). \\ \quad \Pi_{a : \widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]})} \Sigma\left(n : F^{\widehat{\mathcal{E}B}}(c^{j-1}_{[\mathsf{m}-1]}, \mathsf{proj}(w(p,a)))\right).\, F^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}, (w(p,a), n)) \\ \qquad\qquad \Big\downarrow \\ \qquad\qquad \mathsf{lim}_{\left\{\, x \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) \;\middle|\; x \prec c^j_{[\mathsf{m}]}, x \not\equiv c^{j-1}_{[\mathsf{m}-1]} \,\right\}} \widehat{\mathcal{N}} \end{array} \tag{63}$$

is also a homotopy equivalence. The lemma is therefore shown if we can prove that the domain of the above fibration (63), a rather lengthy expression, is homotopy equivalent to $\widehat{\mathcal{N}}(c^j_{[\mathsf{m}]})$. Our first step is to apply the distributivity law (2) to transform this expression to

$$\begin{aligned}
&\Sigma\big(p : \mathsf{lim}_{\left\{\, x \in (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) \;\middle|\; x \prec c^j_{[\mathsf{m}]}, x \not\equiv c^{j-1}_{[\mathsf{m}-1]} \,\right\}} \widehat{\mathcal{N}}\big).\\
&\Sigma\big(n : \Pi_{a:\widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]})} F^{\widehat{\mathcal{E}B}}(c^{j-1}_{[\mathsf{m}-1]}, \mathsf{proj}(w(p,a)))\big).\\
&\quad \Pi_{a:\widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]})} F^{\widehat{\mathcal{E}B}}(c^j_{[\mathsf{m}]}, (w(p,a), n(a))).
\end{aligned} \tag{64}$$

When we look at the following square, in which w is the map (57), w' is induced by the natural transformation v in the same way as w, and proj, proj' come from the restriction to subcategories,

$$\begin{array}{ccc}
\big(\mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!) - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{N}} \circ U\big) \times \widehat{\mathcal{T}}A(c^j_{[\mathsf{m}]}) & \xrightarrow{\;w\;} & \mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+ - c^{j-1}_{[\mathsf{m}-1]}} \widehat{\mathcal{E}B} \circ U \\
\Big\downarrow \mathsf{proj}' & & \Big\downarrow \mathsf{proj} \\
\big(\mathsf{lim}_{c^{j-1}_{[\mathsf{m}-1]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)} \widehat{\mathcal{N}} \circ U\big) \times \widehat{\mathcal{T}}A(c^{j-1}_{[\mathsf{m}-1]}) & \xrightarrow{\;w'\;} & \mathsf{lim}_{c^{j-1}_{[\mathsf{m}-1]} /\!\!/ \widehat{\Delta}^{\mathrm{op}}_+} \widehat{\mathcal{E}B} \circ U
\end{array} \tag{65}$$

we can see that it commutes due to the naturality of the natural transformation v. In particular, note that $\widehat{\mathcal{T}}A$ maps the single morphism $c^j_{[\mathsf{m}]} \to c^{j-1}_{[\mathsf{m}-1]}$ to the identity on A^{m+1-j}. This is exactly what is needed to see that the second line of (64) corresponds to the "missing Σ-component" $\widehat{\mathcal{N}}(c^{j-1}_{[\mathsf{m}-1]})$ in the limit of the first line. Hence, the first and the second line can be "merged" and are equivalent to $\mathsf{lim}_{c^j_{[\mathsf{m}]} /\!\!/ (\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)} \widehat{\mathcal{N}} \circ U$, in other words, $M^{\widehat{\mathcal{N}}}(c^j_{[\mathsf{m}]})$. Comparing the third line of (64) with the definition of the "fibrant diagram of natural transformations" (see (39)), we see that (64) is indeed equivalent to $\widehat{\mathcal{N}}(c^j_{[\mathsf{m}]})$, as required. ◀

By pullback (Lemma 11 and preservation of homotopy equivalences along pullbacks), we immediately get:

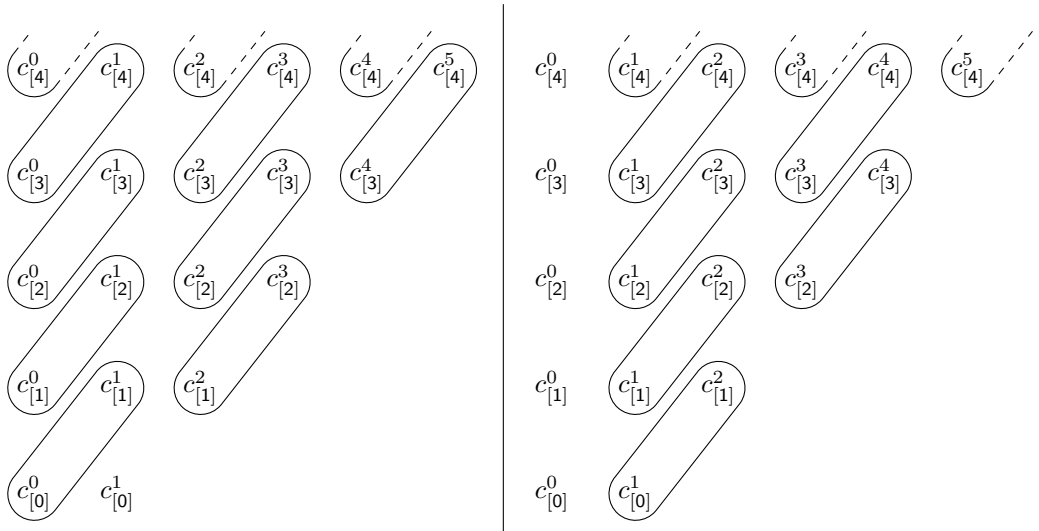
▶ **Corollary 24.** *Let D be a downwards closed subcategory of $\widehat{\Delta}^{\mathrm{op}}_+$ which does not contain the objects $c^j_{[\mathsf{m}]}$ and $c^{j-1}_{[\mathsf{m}-1]}$, but all other predecessors of $c^j_{[\mathsf{m}]}$. The full subcategory of $\widehat{\Delta}^{\mathrm{op}}_+$ which has all the objects of D and the objects $c^{j-1}_{[\mathsf{m}-1]}$, $c^j_{[\mathsf{m}]}$ (for which we write $D + c^{j-1}_{[\mathsf{m}-1]} + c^j_{[\mathsf{m}]}$) is also downwards closed and the fibration*

$$\mathsf{lim}_{D + c^{j-1}_{[\mathsf{m}-1]} + c^j_{[\mathsf{m}]}} \widehat{\mathcal{N}} \;\twoheadrightarrow\; \mathsf{lim}_D \widehat{\mathcal{N}} \tag{66}$$

is a homotopy equivalence. ◀

Corollary 24 is the crucial statement that summarises all of our efforts so far. We can use it to "add and remove" contractible Σ-components in the same way as we did it in the motivating examples (Section 2). More precisely, we exploit that we can group together components of $\widehat{\Delta}^{\mathrm{op}}_+$ in two different ways. Our main lemma is the following:

▶ **Main Lemma 25.** *Given types A, B, recall that we have defined $s : B \to (A \xrightarrow{\omega} B)$ in Definition 20. Assume further that we are given a point $\mathfrak{a}_\circ : A$ (i.e. we regard s as a morphism in $(\mathfrak{C}/A)_{\mathsf{f}}$ instead of $\mathfrak{C}$). Then, the function s is a homotopy equivalence.*



(a) The sequence $D_0 \subset D_1 \subset D_2 \subset \dots$ starts with $\{c^1_{[0]}\}$

(b) The sequence $D'_0 \subset D'_1 \subset D'_2 \subset \dots$ starts with $\{c^0_{[\mathsf{n}]}\}$

Figure 7 Two infinite sequences of downwards closed full subcategories of $(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)$, constructed in the proof of Main Lemma 25: the first starts with $D_0 :\equiv \{c^1_{[0]}\}$. In each step, exactly two objects are added, paired as shown in the left drawing. The second sequence starts with $D'_0 :\equiv \{c^0_{[\mathsf{n}]}\}$ (the leftmost column), i.e. D'_0 is isomorphic to ω^{op}. The pairings are shown in the right drawing. The reader who has read through the proof of Main Lemma 25 is invited to combine the current figure with Figure 4 in order to reconstruct the proof of Proposition 3.

Proof. Using the point $\mathfrak{a}_\circ$, we define $\widehat{\mathcal{N}}$ and $\overline{s} : B \to \mathsf{lim}_{(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)}\widehat{\mathcal{N}}$ as before in (52), and consider the following:

$$\begin{array}{ccccccc} B & \xrightarrow{\overline{s}} & \mathsf{lim}_{(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)}\widehat{\mathcal{N}} & \xrightarrow{\mathsf{pr}'} & \widehat{\mathcal{N}}(c^1_{[0]}) & \xrightarrow{\sim} & B \\ & \searrow^{s} & \downarrow{\scriptstyle \mathsf{pr}} & & & & \\ & & A & \xrightarrow{\omega} & B & & \end{array} \tag{67}$$

The commutativity of the triangle on the left is given by (53). Our first goal is to show that the fibration pr' is a homotopy equivalence.

Consider the set $S :\equiv \left\{ (m,j) \in \mathbb{N}^2 \;\middle|\; j \text{ is even and } j \leq m+1 \right\}$. A pair (m,j) is in S if and only if $c^j_{[\mathsf{m}]}$ is an object in an "odd column" of $\widehat{\Delta}_+^{\mathrm{op}}$ in Figure 3 on page 131 (where we consider the leftmost column the "first"). Define a total order on S by letting (k,i) be smaller than (m,j) if either $k+i < m+j$ or ($k+i \equiv m+j$ and $i<j$). We represent this total order by an isomorphism $f : \mathbb{N}_+ \to S$ (where $\mathbb{N}_+$ are the positive natural numbers) which has the property that $f(n)$ is always smaller than $f(n+1)$. Write $f_1(n)$ and $f_2(n)$ for the first respectively the second projection of $f(n)$.

Let us define a sequence $D_0 \subset D_1 \subset D_2 \subset D_3 \subset \dots$ of full subcategories of $(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)$ by

$$D_0 :\equiv \{c^1_{[0]}\} \tag{68}$$

$$D_n :\equiv D_{n-1} + c^{f_2(n)}_{[\mathsf{f_1(n)}]} + c^{f_2(n)+1}_{[\mathsf{f_1(n)+1}]}. \tag{69}$$

This construction illustrated in Figure 7a. It is easy to see that every object $c^j_{[\mathsf{m}]}$ is added

exactly once, i.e. it is either $c^1_{[0]}$ or it is of the form $c^{f_2(n)}_{[\mathsf{f}_1(\mathsf{n})]}$ or of the form $c^{f_2(n)+1}_{[\mathsf{f}_1(\mathsf{n})+1]}$ for exactly one n. We have chosen the total order on S in such a way that every D_n is a downwards closed full subcategory of $(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)$. Applying Corollary 24, we get a sequence

$$\mathsf{lim}_{D_0}\widehat{\mathcal{N}} \xleftarrow{\sim} \mathsf{lim}_{D_1}\widehat{\mathcal{N}} \xleftarrow{\sim} \mathsf{lim}_{D_2}\widehat{\mathcal{N}} \xleftarrow{\sim} \mathsf{lim}_{D_3}\widehat{\mathcal{N}} \xleftarrow{\sim} \dots \tag{70}$$

of acyclic fibrations. Lemma 7 then shows that the canonical map

$$\mathsf{lim}_{\widehat{\Delta}^{\mathrm{op}}_+}\widehat{\mathcal{N}} \twoheadrightarrow \mathsf{lim}_{D_0}\widehat{\mathcal{N}} \tag{71}$$

is an acyclic fibration. As $\mathsf{lim}_{D_0}\widehat{\mathcal{N}}$ is simply $\widehat{\mathcal{N}}(c^1_{[0]})$, this proves that pr' is indeed a homotopy equivalence.

Next, we want to show the same about pr. We proceed very similarly. This time, we define $S' :\equiv \left\{ (m,j) \in \mathbb{N}^2 \;\middle|\; j \text{ is odd and } j \leq m+1 \right\}$. A pair (m,j) is consequently in S' if and only if $c^j_{[\mathsf{m}]}$ is an object in an "even" column of Figure 4. As before, we define an isomorphism $f' : \mathbb{N}_+ \to S'$, and define a sequence $D'_0 \subset D'_1 \subset D'_2 \subset D'_3 \subset \dots$ of full subcategories of $(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)$ by

$$D'_0 :\equiv \{c^0_{[\mathsf{m}]}\} \qquad \text{(i.e. the full subcategory corresponding to } (\!|\Delta^{\mathrm{op}}_+|\!)) \tag{72}$$

$$D'_n :\equiv D'_{n-1} \;+\; c^{f_2(n)}_{[\mathsf{f}_1(\mathsf{n})]} \;+\; c^{f_2(n)+1}_{[\mathsf{f}_1(\mathsf{n})+1]}. \tag{73}$$

We illustrate the construction of this sequence in Figure 7b, which the reader is encourage to compare with Figure 7a. Again, every object $c^j_{[\mathsf{m}]}$ is added exactly once, and every D_n is downwards closed. Corollary 24 and Lemma 7 then tell us that $\mathsf{lim}_{(\!|\widehat{\Delta}^{\mathrm{op}}_+|\!)}\widehat{\mathcal{N}} \twoheadrightarrow \mathsf{lim}_{\{c^0_{[\mathsf{m}]}\}}\widehat{\mathcal{N}}$ is an acyclic fibration. Hence, pr is indeed a homotopy equivalence, as claimed.

We take another look at the diagram (67). The composition of the three horizontal arrows is the identity by (54). But homotopy equivalences satisfy "2-out-of-3", and we can conclude that $\overline{s}$ is an equivalence. Using "2-out-of-3" again, we see that s is an equivalence as well. ◀

It is straightforward to define what it mean for a type-theoretic fibration category to have propositional truncations by imitating the characterisation given in Section 2. We now show:

▶ **Lemma 26.** *If $\mathfrak{C}$ has propositional truncations, then $\|A\|$ implies that the canonical function $s : B \to (A \xrightarrow{\omega} B)$, viewed as a morphism in $\mathfrak{C}$, is a homotopy equivalence. More precisely, we can construct a function*

$$\|A\| \to \mathsf{isequiv}(s) \tag{74}$$

in $\mathfrak{C}$.

Proof. We have shown in Main Lemma 25 that s is a homotopy equivalence in $(\mathfrak{C}/A)_{\mathsf{f}}$, i.e. if we pull back its domain and codomain along $A \twoheadrightarrow \mathbf{1}$. In $\mathfrak{C}$, this means that

$$\lambda(a,b).(a,s(b)) : A \times B \;\to\; A \times (A \xrightarrow{\omega} B) \tag{75}$$

is an equivalence, but this implies

$$A \to \mathsf{isequiv}(s). \tag{76}$$

The claim then follows from the ordinary universal property of the propositional truncation. ◀

This allows us to prove our main result:

▶ **Theorem 27** (General universal property of the propositional truncation). *Let $\mathfrak{C}$ be a type-theoretic fibration category that satisfies function extensionality, has propositional truncations, and Reedy ω^{op}-limits. Let A and B be two types, i.e. objects in $\mathfrak{C}$. Using the canonical function $s : B \to (A \xrightarrow{\omega} B)$ as defined in Definition 20, we can construct a function*

$$(\|A\| \to B) \;\to\; (A \xrightarrow{\omega} B), \tag{77}$$

and this function is a homotopy equivalence.

Proof. From Lemma 26 we can conclude, just as in the special cases in Section 2, that

$$(\|A\| \to B) \;\to\; \big(\|A\| \to (A \xrightarrow{\omega} B)\big) \tag{78}$$

$$f \;\mapsto\; \lambda x.s(f(x)) \tag{79}$$

is a homotopy equivalence.

This is not yet what we aim for. We need a statement corresponding to the infinite case of Lemma 1, i.e. we need to prove that $\|A\| \to (A \xrightarrow{\omega} B)$ is equivalent to $A \xrightarrow{\omega} B$. To do this, we consider the diagram $\mathcal{P} : (\!|\Delta_+^{\mathrm{op}}|\!) \to \mathfrak{C}$, defined on objects by

$$\mathcal{P}_{[\mathsf{k}]} :\equiv \|A\| \to \mathcal{N}_{[\mathsf{k}]}, \tag{80}$$

and on morphisms by

$$\mathcal{P}(g) :\equiv \lambda(h : \|A\| \to \mathcal{N}_{[\mathsf{k}]}).\lambda x.\mathcal{N}(g)(h(x)). \tag{81}$$

Paolo Capriotti has pointed out that $\mathcal{P}$ is Reedy fibrant, and this is a crucial observation. As $\mathcal{P}$ is defined over a poset, it is enough to show that (81) is a fibration for every g. Our argument is the following: The maps in both directions which are used to prove the distributivity law (2) are *strict* inverses, i.e. their compositions (in both orders) are judgmentally equal to the identities. This means that every $\mathcal{P}_i$ is isomorphic to a Σ-type, where we "distribute" $\|A\|$ over the Σ-components. From this representation, it is clear that $\mathcal{P}(g)$ is always a fibration, as fibrations are closed under composition with isomorphisms.

Because of Lemma 1 (and the fact that the equivalence there can be defined uniformly), there is a natural transformation $\kappa : \mathcal{P} \to \mathcal{N}$ which is levelwise a homotopy equivalence. By the definition of $\mathfrak{C}$ having Reedy ω^{op}-limits, the resulting arrow between the two limits, that is

$$\mathsf{lim}_{(\!|\Delta_+^{\mathrm{op}}|\!)}(\kappa) : \big(\|A\| \to (A \xrightarrow{\omega} B)\big) \;\to\; (A \xrightarrow{\omega} B), \tag{82}$$

is a homotopy equivalence as well. To conclude, we simply compose (79) and (82). ◀

10 Finite Cases

If B is an n-type for some finite fixed number n, the higher coherence conditions should intuitively become trivial. This is obvious for the representation of $\mathcal{N}$ and $\mathcal{E}B$ given in Figures 2 and 4, although admittedly not for our actual definition of $\mathcal{E}B$ in Section 5 (and the corresponding definition of $\mathcal{N}$ and $\widehat{\mathcal{N}}$) where it requires a little more thought. This is our main goal for this section. For the presentation, we assume that the type theory in question has a universe $\mathcal{U}$, although this assumption is not strictly necessary. After this, it will be easy to see that the universal properties of the propositional truncation with an n-type as codomain, for any externally fixed number n, can be formulated and proved in standard syntactical homotopy type theory.

We start by reversing the statement that "singletons are contractible":

▶ **Lemma 28.** *Assume A is a type, $B : A \to \mathcal{U}$ a family, and $C : (\Sigma\,(a : A)\,.\,B(a)) \to \mathcal{U}$ a second family. The following are logically equivalent:*

(i) *For any $a : A$, there is a point $b_a : B(a)$ and a homotopy equivalence*

$$C(a,b) \simeq (b = b_a). \tag{83}$$

(ii) *The canonical projection*

$$\mathsf{fst} : \big(\Sigma\,(a : A)\,.\,\Sigma\,(b : B(a))\,.\,C(a,b)\big) \to A \tag{84}$$

is a homotopy equivalence.

Proof. The direction (1)⇒(2) is an obvious consequence from the contractibility of singletons. For the other direction, recall that, for any type X and families $Y, Z : X \to \mathcal{U}$, a map

$$f : \Pi_{x:X}\,(Y(x) \to Z(x)) \tag{85}$$

is a *fibrewise (homotopy) equivalence* if each $f(x) : Y(x) \to Z(x)$ is a homotopy equivalence [27, Chapter 4.7]. Given (85), there is a canonical way to define a map on the total spaces

$$\mathsf{total}(f) : \Sigma\,(x : X)\,.\,Y(x) \;\to\; \Sigma\,(x : X)\,.\,Z(x). \tag{86}$$

Then, $\mathsf{total}(f)$ is a homotopy equivalence if an only if f is a fibrewise homotopy equivalence [27, Thm. 4.7.7]. Using this result, we derive a very short proof of (2)⇒(1):

We fix $a : A$ and assume (2) which implies that $\Sigma\,(b : B(a))\,.\,C(a,b)$ is contractible. This gives us the required b_a and allows us to define a map

$$g : \Pi_{b:B(a)}\,(C(a,b) \to b_a = b)\,. \tag{87}$$

Clearly, $\mathsf{total}(g)$ is a homotopy equivalence as it is a map between contractible types. Hence, g is a fibrewise homotopy equivalence. ◀

We are now ready to show that, in the case of n-types, the higher "fillers for complete boundaries" become homotopically simpler and simpler, and finally trivial.

▶ **Lemma 29.** *Let $n \geq -2$ be a number and B be a type in $\mathfrak{C}$. Consider the equality semi-simplicial type $\mathcal{E}B : \Delta_+^{\mathrm{op}} \to \mathfrak{C}$ of B. For an object $[\mathsf{k}]$ of Δ_+^{op}, we can consider the fibration $\mathcal{E}B_{[\mathsf{k}]} \twoheadrightarrow M^{\mathcal{E}B}_{[\mathsf{k}]}$. We know that, by definition, the fibre over $m : M^{\mathcal{E}B}_{[\mathsf{k}]}$ is simply $\Sigma\,(x : B)\,.\,\tilde{\eta}_{[\mathsf{k}]}(x) = m$.*

If B is an n-type, then, for any object $[\mathsf{k}]$ of Δ_+^{op}, all these fibres are $(n-k)$-truncated (or contractible, if this difference is below -2).

▶ Remark. The other direction of Lemma 29 should also hold, as $M^{\mathcal{E}B}_{[\mathsf{k}]}$ should be equivalent to $\Sigma\,(b : B)\,.\,\Omega^k(B,b)$. We do neither prove nor require this direction here.

Proof of Lemma 29. The statement clearly holds for $[\mathsf{k}] \equiv [\mathsf{0}]$, as the matching object $M^{\mathcal{E}B}_{[\mathsf{k}]}$ will in this case be the unit type. We assume that the statement holds for $[\mathsf{k}]$ and show it for $[\mathsf{k}+\mathsf{1}]$. Recall our notation from Section 6 (see right before Main Lemma 15): If s is some set, we write $\overline{s}$ for the poset generated by s. If i is an element of s, then $\overline{s}_{-i}$ is the poset $\overline{s}$ without the set s and without the set $s - i$.

Consider the following diagram in the poset $\mathsf{Sub}([\mathsf{k}+\mathsf{1}])$:

$$\begin{array}{ccc} \overline{[\mathsf{k}+1]} - [\mathsf{k}+1] & \longrightarrow & \overline{[\mathsf{k}]} \\ \downarrow & & \downarrow \\ \overline{[\mathsf{k}+1]}_{-0} & \longrightarrow & \overline{[\mathsf{k}]} - [\mathsf{k}] \end{array}$$

If we apply the functor $\mathsf{lim}_{\_}(\mathcal{E}B \circ U)$ on this square, we get

$$\begin{array}{ccc} M^{\mathcal{E}B}_{[\mathsf{k}+1]} & \twoheadrightarrow & \mathcal{E}B_{[\mathsf{k}]} \\ \downarrow & & \downarrow \\ \mathsf{lim}_{\overline{[\mathsf{k}+1]}_{-0}}(\mathcal{E}B \circ U) & \twoheadrightarrow & M^{\mathcal{E}B}_{[\mathsf{k}]} \end{array}$$

where the bottom left type is the 0-th $[\mathsf{k}]$-horn as in Main Lemma 15. By the induction hypothesis, the right vertical fibration is an $(n-k)$-truncated type. By Lemma 11, the square is a pullback. This means that the left vertical fibration is $(n-k)$-truncated as well, as fibres on the left side are homotopy equivalent to fibres on the right side.

Consider the composition of fibrations

$$\mathcal{E}B_{[\mathsf{k}+1]} \twoheadrightarrow M^{\mathcal{E}B}_{[\mathsf{k}+1]} \twoheadrightarrow \mathsf{lim}_{\overline{[\mathsf{k}+1]}_{-0}}(\mathcal{E}B \circ U). \tag{88}$$

Intuitively, the horn is a "tetrahedron with missing filler and one missing face", the matching object is the same plus one component which represents this face, and $\mathcal{E}B_{[\mathsf{k}]}$ has, in addition to the face, also a filler of the whole boundary. The filler is really the statement that the "new" face equals the canonical one, and we can now make this intuition precise by applying Lemma 28. Let us check the conditions:

- Certainly, we can write the sequence in the form

$$\Sigma\,(x : X)\,.\,\Sigma\,(x : Y(x))\,.\,Z(x,y) \twoheadrightarrow \Sigma\,(x : X)\,.\,Y(x) \twoheadrightarrow X \tag{89}$$

  (this is given by Lemma 12).
- The composition is a homotopy equivalence by Main Lemma 15.

Thus, we can assume that $Z(x,y)$ is equivalent to $y =_{Y(x)} y_x$ for some y_x, and thereby of a truncation level that is by one lower than $Y(x)$. But the latter is $(n-k)$ as we have seen above.[2] ◀

As a corollary, we get the case for $[\mathsf{k}] \equiv [\mathsf{n}+2]$:

▶ **Corollary 30.** *Let B be an n-type. Then, the fibration*

$$\mathcal{E}B_{[\mathsf{n}+2]} \twoheadrightarrow M^{\mathcal{E}B}_{[\mathsf{n}+2]} \tag{90}$$

is a homotopy equivalence. ◀

We are now in the position to formulate our result for n-types with finite n. Recall from Definition 19 that we write $A \xrightarrow{[\mathsf{n}]} B$ for $\mathcal{N}_n$.

[2] On low levels, we can consider the situation in terms of the presentation in Figure 2. Here, y_x will be the "missing face" that one gets by gluing together the other faces.

▶ **Theorem 31** (Finite general universal property of the propositional truncation)**.** *Let n be a fixed number, $-2 \leq n < \infty$. In Martin-Löf type theory with propositional truncations and function extensionality we can, for any type A and any n-type B, derive a canonical function*

$$\big(\|A\| \to B\big) \;\to\; \big(A \xrightarrow{[n+1]} B\big) \tag{91}$$

that is a homotopy equivalence.

Proof. Looking at Corollary 30 and at the definition of $\mathcal{N}$, as given in Section 7, we see immediately that each $\mathcal{N}_{[k+1]} \twoheadrightarrow \mathcal{N}_{[k]}$ with $k \geq n+1$ is a homotopy equivalence. Thus, using Lemma 7, the Reedy limit $\mathsf{lim}_{(\!|\Delta_+^{\mathrm{op}}|\!)}\mathcal{N}$ is equivalent to $\mathcal{N}_{[n+1]}$, and these are $A \xrightarrow{\omega} B$ and $A \xrightarrow{[n+1]} B$ by definition. Similarly, the limit $\mathsf{lim}_{(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)}\widehat{\mathcal{N}}$ (which we used in the proof of Main Lemma 25) is homotopy equivalent to the limit over $(\!|\widehat{\Delta}_+^{\mathrm{op}}|\!)$ restricted to $\left\{ c^i_{[k]} \;\middle|\; k \leq n+1 \right\}$. It is easy to see that the whole proof can be carried out using only finite parts of the infinite diagrams. But then, of course, all we need are finitely many nested Σ-types instead of Reedy ω^{op}-limits, and these automatically exist. Further, the only point where we crucially used the judgmental η-rule for Σ is the proof of Theorem 27. In the finite case, however, this is not necessary, as Lemma 1 is sufficient (similarly, the judgmental η-rule for Π-types is not necessary). Therefore, the whole proof can be carried out in the standard version of MLTT with propositional truncations. ◀

11 Concluding Remarks

For any type B, we have constructed the equality semi-simplicial type $\mathcal{E}B : \Delta_+^{\mathrm{op}} \to \mathfrak{C}$, and we have shown that natural transformations from the *trivial* diagram $\mathcal{T}A$ (the $[0]$-coskeleton of the diagram constantly A) to $\mathcal{E}B$ correspond to maps $\|A\| \to B$. The construction required us to assume that $\mathfrak{C}$ has Reedy ω^{op}-limits. There are several points that we would like to discuss briefly here, all of which naturally raise further open questions.

First, there are many connections to constructions and results in homotopy theory and the theory of higher topoi, model categories, and quasi-categories. As we have discussed, for any type B and any inverse category $\mathfrak{I}$ that is admissible for $\mathfrak{C}$, the constructed equality diagram $\mathcal{E}B : \mathfrak{I} \to \mathfrak{C}$ is a Reedy fibrant replacement of the diagram that is constantly B. Similarly, the diagram $\mathcal{T}A$ is a $[0]$-coskeleton. One anonymous reviewer has pointed out that Theorem 27 is a type-theoretic version of a result on $(\infty, 1)$-topoi by Lurie [18, Proposition 6.2.3.4]. There are certainly deep connections that have yet to be explored.

Second, we have presented the assumptions of Reedy ω^{op}-limits as a necessary requirement. However, we are not aware of a model in which the necessary limits are absent. Even though it seems very unlikely, it is in principle possible that these Reedy limits exist in any type-theoretic fibration category automatically. Assume A and B are some given types. We do not know whether it is possible to define the expression $\mathcal{N}_{A,B}(n)$ for a *variable* n in HoTT, i.e. to give a function $f_{A,B} : \mathbb{N} \to \mathcal{U}$ (where $\mathcal{U}$ is a universe) such that the type $f_{A,B}(n)$ is equivalent to $\mathcal{N}_{A,B}(n)$ for all *numerals* n.

If this can be done, it should be possible to actually construct what is intuitively an "infinite Σ-type", by asking for all finite approximations with proofs that they fit together, and we could reasonably hope that Theorem 27 can be proved in HoTT without any further assumptions. This has been made precise for the more general case of M-types by Ahrens, Capriotti and Spadotti [1]. However, we do not expect that such a function $f_{A,B}$ can be defined. This is at least as hard as defining the equality semi-simplicial type over B as a

function $\mathcal{E}B : \mathbb{N} \to \mathcal{U}$; this would correspond to the special case where A is the unit type. Defining $\mathcal{E}B$ in this way, however, seems to be as difficult as the famous open problem of defining semi-simplicial types internally as a function $\mathsf{SS} : \mathbb{N} \to \mathcal{U}_1$ (where $\mathcal{U}_1$ is a universe that is larger than $\mathcal{U}$). The two problems are identical apart from the fact that the fibres over the matching objects differ. For $\mathcal{E}B$, the fibre over a point m of the matching object is $\Sigma\,(x : B)\,.\,\tilde{\eta}_i(x) = m$ as can be seen from (23) on page 125, while for SS, the fibres are constantly the universe $\mathcal{U}$. The author does not expect that this makes a real difference in difficulty. It seems likely that a function $\mathcal{E}B$ would enable us to talk about coherent equalities so that we could define the function SS, implying that defining $\mathcal{E}B$ is at least not easier.

Going back a step, while we can prove Theorem 31 internally if n is instantiated with any numeral, we conjecture that it is impossible to prove it for a variable n. What we think is certainly possible is to write a program in any standard programming language that takes a number n as input and prints out the formalised statement of Theorem 31 (in the syntax of a proof assistant such as Coq or Agda) together with a proof. Even in Agda itself, we would be able to define a function which generates the Agda source code of Theorem 31, for any natural number n. This would provide a solution if we were able to interpret syntax of HoTT in HoTT, which is another famous open problem [23].

Third, instead of asking whether HoTT allows us to define Reedy fibrant diagrams such as $\mathcal{E}B$ or SS, we may choose to work in a theory in which we know that it is possible. Candidates are Voevodsky's HTS (homotopy type system) [29], or the two-level system outlined by Altenkirch, Capriotti and the current author [2]. We believe that the results of the current article can be formalised in such settings.

Fourth, is seems obvious to ask whether statements analogous to Theorems 27 and 31 can be derived for higher truncation operators, written $\|-\|_n$ [27, Chapter 7.3]. A partial result, namely a characterisation of maps $\|A\|_k \to B$ if B is $(k+1)$-truncated, have been obtained by Capriotti, Vezzosi and the current author [6].

More general results are currently unknown, but we want to conclude with a conjecture. Assume a type A and an object $[\mathsf{k}]$ of Δ_+ are given. We define the *(fibrant)* $[\mathsf{k}]$*-skeleton* of the diagram that is constantly A, written $\mathsf{coskel}^{[\mathsf{k}],A}$, by giving the fibres over the matching objects:

$$\mathsf{coskel}^{[\mathsf{k}],A}_{[\mathsf{i}]} :\equiv \begin{cases} \mathcal{E}A_{[\mathsf{i}]} & \text{if } [\mathsf{i}] \preceq [\mathsf{k}] \\ M^{\mathsf{coskel}^{[\mathsf{k}],A}}_{[\mathsf{i}]} & \text{else.} \end{cases} \tag{92}$$

Note that, with this definition, the diagram $\mathcal{T}A$ that we have defined earlier is not *exactly* $\mathsf{coskel}^{[0],A}$ for the same reason as for which $\mathcal{E}A_{[0]}$ is not *exactly* A, but of course, $\mathsf{coskel}^{[0],A}$ and $\mathcal{T}A$ are homotopy equivalent. In principle, we could have done the whole proof with $\mathsf{coskel}^{[0],A}$ instead of $\mathcal{T}A$. Merely for convenience, we have taken advantage of the fact that $\mathcal{T}A$ is already Reedy fibrant.

For a number $n \geq -1$, we conjecture that natural transformations from $\mathsf{coskel}^{[\mathsf{n}+1],A}$ to $\mathcal{E}B$ correspond to functions $\|A\|_n \to B$. Even more generally, given a higher inductive type H, it may be possible to determine a representation of H as a diagram $\mathsf{Rep}(H) : \Delta_+^{\mathrm{op}} \to \mathfrak{C}$ such that natural transformations from $\mathsf{Rep}(H)$ to $\mathcal{E}B$ corresponds to functions $H \to B$. This is very simple for non-recursive higher "inductive" types that do not refer to refl or applications of J in their constructors: for example, the circle $\mathbb{S}^1$ can be represented with $\mathsf{Rep}(\mathbb{S}^1)_{[0]} \equiv \mathsf{Rep}(\mathbb{S}^1)_{[1]} \equiv \mathbf{1}$ and $\mathsf{Rep}(\mathbb{S}^1)_{[\mathsf{n}+2]} \equiv \mathbf{0}$, while the suspension of A can be realised as $\mathsf{Rep}(\Sigma A)_{[0]} \equiv \mathbf{2}$, $\mathsf{Rep}(\Sigma A)_{[1]} \equiv A$, and $\mathsf{Rep}(\Sigma A)_{[\mathsf{n}+2]} \equiv \mathbf{0}$. If this turns out to work for a larger class of higher inductive types, it may be understood as a type-theoretic version

of the *homotopy hypothesis* which has so far suffered from the difficulty of formulating the coherences of categorical laws [11].

Acknowledgements. First of all, I want to thank Paolo Capriotti. It was him who pointed out to me that what I was doing could be formulated nicely in terms of diagrams over inverse categories, and he helped me to transfer my original definition of the equality semi-simplicial type to this setting. Numerous discussions with him helped me greatly to understand Shulman's work and related concepts. He, as well as Christian Sattler and the anonymous reviewers, has enabled me to understand connections with homotopy theory that I had not been aware of. Especially the reviewers' reports have been helpful for the improvements that have been incorporated in this article. Steve Awodey, as examiner of my Ph.D. thesis, has given me very useful feedback on this work as well.

I further want to thank Thorsten Altenkirch for his constant general support, Martín Escardó for many discussions on related questions, and Michael Shulman for making his LaTeX macros publicly available.

References

1 Benedikt Ahrens, Paolo Capriotti, and Régis Spadotti. Non-wellfounded trees in homotopy type theory. In *Typed Lambda Calculi and Applications (TLCA)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 17–30. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

2 Thorsten Altenkirch, Paolo Capriotti, and Nicolai Kraus. Infinite structures in type theory: Problems and approaches. Presented at TYPES, Tallinn, Estonia, 20 May 2015. Work in progress, working title.

3 Steve Awodey. Natural models of homotopy type theory. *arXiv preprints*, arXiv:1406.3219, Jun 2014.

4 Steve Awodey and Andrej Bauer. Propositions as [types]. *Journal of Logic and Computation*, 14(4):447–471, 2004.

5 Michael Batanin. Monoidal globular categories as a natural environment for the theory of weak n-categories. *Advances in Mathematics*, 136(1):39–103, 1998.

6 Paolo Capriotti, Nicolai Kraus, and Andrea Vezzosi. Functions out of higher truncations, 2015. In preparation, to appear in *Computer Science Logic (CSL)*.

7 R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the NuPRL Proof Development System*. Prentice-Hall, NJ, 1986.

8 Nicola Gambino and Richard Garner. The identity type weak factorisation system. *Theoretical Computer Science*, 409:94–109, Dec 2008.

9 Hugo Herbelin. A dependently-typed construction of semi-simplicial types. *Mathematical Structures in Computer Science*, pages 1–16, Mar 2015.

10 Jason J. Hickey. Formal objects in type theory using very dependent types. In *Foundations of Object Oriented Languages 3*, 1996.

11 André Hirschowitz, Tom Hirschowitz, and Nicolas Tabareau. Wild omega-categories for the homotopy hypothesis in type theory. In *Typed Lambda Calculi and Applications (TLCA)*, volume 38 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 226–240. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2015.

12 Mark Hovey. *Model Categories*. Number 63 in Mathematical surveys and monographs. American Mathematical Society, 2007.

13 Nicolai Kraus. *Truncation Levels in Homotopy Type Theory*. PhD thesis, School of Computer Science, University of Nottingham, Nottingham, UK, 2015.

14 Nicolai Kraus, Martín Escardó, Thierry Coquand, and Thorsten Altenkirch. Notions of anonymous existence in Martin-Löf type theory, 2014. Submitted to the special issue of TLCA'13.

15 Tom Leinster. A survey of definitions of n-category. *Theory and applications of Categories*, 10(1):1–70, 2002.

16 Peter LeFanu Lumsdaine. Weak omega-categories from intensional type theory. In *Typed Lambda Calculi and Applications (TLCA)*, pages 172–187. Springer-Verlag, 2009.

17 Peter LeFanu Lumsdaine and Michael A. Warren. The local universes model: an overlooked coherence construction for dependent type theories. *Transactions on Computational Logic (TOCL)*, 16(3), 2015. To appear.

18 Jacob Lurie. *Higher topos theory*, volume 170 of *Annals of Mathematics Studies*. Princeton University Press, Princeton, NJ, 2009.

19 James E. McClure. On semisimplicial sets satisfying the Kan condition. *Homology, Homotopy and Applications*, 15(1):73–82, 2013.

20 Matt Oliveri. A formalized interpreter. Blog post, `http://homotopytypetheory.org/2014/08/19/a-formalized-interpreter/`.

21 Colin Patrick Rourke and Brian Joseph Sanderson. Δ-sets I: Homotopy theory. *The Quarterly Journal of Mathematics*, 22(3):321–338, 1971.

22 Graeme Segal. Classifying spaces and spectral sequences. *Publications Mathématiques de l'Institut des Hautes Études Scientifiques*, 34(1):105–112, 1968.

23 Michael Shulman. Homotopy type theory should eat itself (but so far, it's too big to swallow). Blog post, homotopytypetheory.org/2014/03/03/hott-should-eat-itself.

24 Michael Shulman. Univalence for inverse diagrams and homotopy canonicity. *Mathematical Structures in Computer Science*, pages 1–75, Jan 2015.

25 Ross Street. The algebra of oriented simplexes. *Journal of Pure and Applied Algebra*, 49(3):283–335, 1987.

26 The HoTT and UF community. Homotopy type theory mailing list, since 2011. Google Groups.

27 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. homotopytypetheory.org/book, Institute for Advanced Study, first edition, 2013.

28 Benno van den Berg and Richard Garner. Types are weak ω-groupoids. *Proceedings of the London Mathematical Society*, 102(2):370–394, 2011.

29 Vladimir Voevodsky. A simple type system with two identity types, 2013. Unpublished note.

On the Structure of Classical Realizability Models of ZF

Jean-Louis Krivine

University Paris-Diderot, CNRS
France
krivine@pps.univ-paris-diderot.fr

Abstract

The technique of *classical realizability* is an extension of the method of *forcing*; it permits to extend the Curry-Howard correspondence between proofs and programs, to Zermelo-Fraenkel set theory and to build new models of ZF, called *realizability models*. The structure of these models is, in general, much more complicated than that of the particular case of forcing models. We show here that the class of constructible sets of any realizability model is an elementary extension of the constructibles of the ground model (a trivial fact in the case of forcing, since these classes are identical). By Shoenfield absoluteness theorem, it follows that every true Σ^1_3 formula is realized by a closed λ_c-term.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases lambda-calculus, Curry-Howard correspondence, set theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.146

1 Introduction

In [6, 7, 9], we have introduced the technique of *classical realizability*, which permits to extend the Curry-Howard correspondence between proofs and programs [5], to Zermelo-Fraenkel set theory. The models of ZF we obtain in this way are called *realizability models*; this technique is an extension of the method of forcing, in which the ordered sets (sets of *conditions*) are replaced with more complex first order structures called *realizability algebras*. These structures are refinements of the well known *combinatory algebras* [3], with the `call/cc` instruction of [4].

We show here that every realizability model $\mathcal{N}$ of ZF contains a transitive submodel, which has the same ordinals as $\mathcal{N}$, and which is an elementary extension of the ground model. It follows that the constructible universe of a realizability model is an elementary extension of the constructible universe of the ground model (a trivial fact in the particular case of forcing, since these classes are *identical*).

We obtain this result by showing the existence of an ultrafilter on the *characteristic Boolean algebra* $\mathbb{B}2$ of the realizability model, which is defined in [7, 9].

From this result, it follows that the *Shoenfield absoluteness theorem* applies to realizability models and therefore that: *Any* Σ^1_3 *formula which is true in the ground model is realized by a closed* λ_c*-term.*

Another application is given in [8]: the *bar-recursion operator* was defined and studied in [1, 2, 10] where it is shown that it realizes the *axiom of dependent choice.*

In [8] it is shown, by means of the results of the present paper, that every closed formula of analysis (i.e. Σ^1_n or Π^1_n) which is true in the ground model, is realized by a closed λ_c-term containing this operator; and that the same is true for the axiom: $\mathbb{R}$ *is well-ordered.*

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 146–161

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

2 Background and notations

We use here the basic notions and notations of the theory of *classical realizability*, which was developed in [6, 7, 9].[1] We consider a model $\mathcal{M}$ of ZF + V = L, which we call the *ground model* [2] and, in $\mathcal{M}$, a *realizability algebra* $\mathcal{A} = (\Lambda, \Pi, \Lambda \star \Pi, \mathrm{QP}, \perp\!\!\!\perp)$. Λ is the set of *terms*, Π is the set of *stacks*, $\Lambda \star \Pi$ is the set of *processes*, $\mathrm{QP} \subset \Lambda$ is the set of *proof-like terms*, and $\perp\!\!\!\perp$ is a distinguished subset of $\Lambda \star \Pi$. They satisfy the axioms of *realizability algebra*, which are given in [6] or [9]. In the model $\mathcal{M}$, we use the language of ZF with the binary relation symbols $\notin, \subset$ and function symbols, which we shall define when needed, by means of formulas of ZF. We can now build (see [6]) the *realizability model* $\mathcal{N}$, which has the same set of individuals as $\mathcal{M}$, the truth value set of which is $\mathcal{P}(\Pi)$, endowed with a suitable Boolean algebra structure (not the usual one for the powerset). The language of this model has three binary relation symbols $\notin, \notin, \subset$, and the same function symbols as the model $\mathcal{M}$, with the same interpretation.

The *formulas* are built as usual, from atomic formulas, *with the only logical symbols* $\perp, \to, \forall$.

ε is called the *strong membership relation*; $\in$ is called the *weak* or *extensional membership relation*.

The formula $\forall z(x \notin z \to y \notin z)$ is written $x = y$; it is the *strong* or *Leibniz equality*. The formula $x \subset y \wedge y \subset x$ is written $x \simeq y$; it is the *weak* or *extensional equality*.

Notations. We shall write:

- $\neg F$ for $F \to \perp$; $F_1, \dots, F_n \to F$ for $F_1 \to (\dots \to (F_n \to F)\dots)$;
- $\exists x\, F$ for $\neg\forall x \neg F$; $\exists x\{F_1, \dots, F_n\}$ for $\neg\forall x(F_1, \dots, F_n \to \perp)$.

We shall often use the notation $\vec{x}$ for a finite sequence $x_1, \dots, x_n$; for instance, we shall write $F[\vec{x}]$ for $F[x_1, \dots, x_n]$.

By means of the completeness theorem, we obtain from $\mathcal{N}$ an ordinary model $\mathcal{N}'$, with truth values in $\{0, 1\}$. The set of individuals of $\mathcal{N}'$ generally *strictly contains* $\mathcal{N}$.

The elements of $\mathcal{N}'$ are called *individuals of* $\mathcal{N}'$ or even *individuals of* $\mathcal{N}$. The individuals are generally denoted by $a, b, c, \dots, a_0, a_1, \dots$

In [6] or [7], we define a theory ZF_ε, written in this language. The axioms for ε are essentially the same as the axioms for $\in$ in ZF (sometimes in an unusual form), *without extensionality*. For instance, the infinity axiom is the following scheme:

$$\forall \vec{z}\forall a\exists b\big\{a\,\varepsilon\, b\,, (\forall x\,\varepsilon\, b)(\exists y\, F[x, y, \vec{z}] \to (\exists y\,\varepsilon\, b)F[x, y, \vec{z}])\big\}$$

for every formula $F[x, y, z_1, \dots, z_n]$.

The axioms for $\in, \subset$ are a kind of coinductive definition from ε:

$$\forall x\forall y(x \in y \leftrightarrow (\exists z\,\varepsilon\, y)x \simeq z) \quad ; \quad \forall x\forall y(x \subset y \leftrightarrow (\forall z\,\varepsilon\, x)z \in y).$$

We show that ZF_ε is a *conservative extension* of ZF, and that the model $\mathcal{N}$ *satisfies the axioms of* ZF_ε, which means that each one of these axioms is *realized by a proof-like term*.

---

[1] The papers [6, 7, 9, 8] are available at `http://www.pps.univ-paris-diderot.fr/~krivine/`.

[2] In fact, it suffices that $\mathcal{M}$ satisfy the *choice principle CP*, which is written as follows, in the language of ZF with a new binary relation symbol $\lhd$: " $\lhd$ *is a well ordering relation on* $\mathcal{M}$ ". It is well known that, in every countable model of ZFC, we can define such a binary symbol, so as to get a model of ZF + CP. Thus, ZF + CP is a *conservative extension* of ZFC.

Given a term $\xi \in \Lambda$ and a closed formula $F[a_1, \ldots, a_n]$ in the language of ZF_ε, with parameters $a_1, \ldots, a_n$ in $\mathcal{N}$ (or, which is the same, in $\mathcal{M}$), we shall write: $\xi \Vdash F[a_1, \ldots, a_n]$ in order to say that the term ξ realizes $F[a_1, \ldots, a_n]$. The truth value of this formula is a subset of Π, denoted by $\|F[a_1, \ldots, a_n]\|$. We write $\Vdash F$ in order to say that F is realized by some proof-like term.

Thus, the model $\mathcal{N}'$ satisfies ZF_ε; therefore, in $\mathcal{N}'$, we can define a model of ZF, denoted $\mathcal{N}'_\in$, in which equality is interpreted by extensional equivalence.

The general properties of the realizability models are described in [9]; we shall use the definitions and notations of this paper.

In what follows, unless otherwise stated, each formula of ZF_ε must be interpreted in $\mathcal{N}$ (its truth value is a subset of Π) or, if one prefers, in $\mathcal{N}'$ (then its truth value is 0 or 1). If the formula must be interpreted in $\mathcal{M}$, (in that case, it does not contains the symbol $\notin$) it will be explicitly stated.

3 Function symbols

Notations. The formula $\forall z(z \notin y \to z \notin x)$ is denoted by $x \subseteq y$ (*strong inclusion*); the formula $x \subseteq y \wedge y \subseteq x$ is denoted by $x \cong y$ (*strong extensional equivalence*). We recall that $\subset$ and $\simeq$ are the symbols of inclusion and of extensional equivalence of ZF: $x \subset y \equiv \forall z(z \notin y \to z \notin x)$; $x \simeq y \equiv (x \subset y \wedge y \subset x)$.

Function symbols associated with axioms of ZF_ε

In this section, we define a function symbol for each of the following axioms of ZF_ε: comprehension, pairing, union, power set and collection.

Comprehension

For each formula $F[y, \vec{z}]$ of ZF_ε, (where $\vec{z}$ is a finite sequence of variables $z_1, \ldots, z_n$) we define, in $\mathcal{M}$, a symbol of function of arity $n+1$, denoted provisionally by $\mathrm{Compr}_F(x, \vec{z})$, (Compr is an abbreviation for *Comprehension*) by setting:

$$\mathrm{Compr}_F(a, \vec{c}) = \{(b, \xi \cdot \pi)\ ;\ (b, \pi) \in a,\ \xi \Vdash F[b, \vec{c}]\}.$$

It was shown in [9] (and it is easily checked) that we have:

$$\|b \notin \mathrm{Compr}_F(a, \vec{c})\| = \|F[b, \vec{c}] \to b \notin a\|\,.$$

Thus, we have:

- $\mathsf{I} \Vdash \forall x \forall y \forall \vec{z}(y \notin \mathrm{Compr}_F(x, \vec{z}) \to (F[y, \vec{z}] \to y \notin x))$;
- $\mathsf{I} \Vdash \forall x \forall y \forall \vec{z}((F[y, \vec{z}] \to y \notin x) \to y \notin \mathrm{Compr}_F(x, \vec{z}))$.

Therefore, instead of $\mathrm{Compr}_F(x, \vec{z})$, we shall use for this function symbol, the more intuitive notation $\{y \varepsilon x\ ;\ F[y, \vec{z}]\}$, in which y is a *bound variable*.

Pairing

We define the following binary function symbol:

$$\mathrm{pair}(x, y) = \{z \varepsilon \{x, y\} \times \Pi\ ;\ (z = x) \vee (z = y)\}.$$

It is easily checked that we have the desired property:

$$\Vdash \forall x \forall y \forall z(z \varepsilon \mathrm{pair}(x, y) \leftrightarrow z = x \vee z = y)\,.$$

▶ Remark. We could also define a symbol $\mathrm{pair}(x,y)$, with this property, directly in $\mathcal{M}$, as follows:

$$\mathrm{pair}(x,y) = \{(x, \underline{1}\cdot\pi)\ ;\ \pi\in\Pi\} \cup \{(y, \underline{0}\cdot\pi)\ ;\ \pi\in\Pi\}.$$

In the sequel, when working in $\mathcal{N}$, we shall use the (natural) abbreviations: $\{x,y\}$ for $\mathrm{pair}(x,y)$; (x,y) for $\mathrm{pair}(\mathrm{pair}(x,x),\mathrm{pair}(x,y))$.

Union and power set

We define below two unary function symbols $\overline{\bigcup}x$ and $\overline{\mathcal{P}}(x)$, such that:

- $\Vdash \forall x\forall z(z\in \overline{\bigcup}x \leftrightarrow (\exists y\in x)\, z\in y)$.
- $\Vdash \forall x(\forall y\in \overline{\mathcal{P}}(x))(\forall z\in y)(z\in x)$; $\Vdash \forall x\forall y(\exists y'\in\overline{\mathcal{P}}(x))\forall z(z\in y'\leftrightarrow z\in x\wedge z\in y)$.

▶ **Theorem 1.** *Let $\mathcal{V},\mathcal{Q}$ be the unary function symbols defined in $\mathcal{M}$ as follows:*

$$\mathcal{V}(a) = \mathrm{Cl}(a)\times\Pi \quad \textit{and} \quad \mathcal{Q}(a) = \mathcal{P}(\mathrm{Cl}(a)\times\Pi)\times\Pi$$

where $\mathrm{Cl}(a)$ is the transitive closure of a. Then, we have:

(i) $\mathsf{I} \Vdash \forall x\forall y\forall z(z\in y, z\notin \mathcal{V}(x)\to y\notin x)$.

(ii) $\mathsf{I} \Vdash \forall x\forall\vec{z}(\{y\in x\ ;\ F[y,\vec{z}]\}\in\mathcal{Q}(x))$ *for every formula $F[x,\vec{z}]$ of $ZF_\in$.*

Proof.

(i) Let a,b,c be individuals in $\mathcal{M}$, $\xi,\eta\in\Lambda$ and $\pi\in\Pi$ such that: $\xi\Vdash c\in b$, $\eta\Vdash c\notin\mathcal{V}(a)$ and $\pi\in\|b\notin a\|$; we have therefore $(b,\pi)\in a$. We must show $\xi\star\eta\cdot\pi\in\bot\!\!\!\bot$. We show that $\|c\notin b\|\subset\|c\notin\mathcal{V}(a)\|$: indeed, if $\rho\in\|c\notin b\|$, then we have $(c,\rho)\in b$. But we have $(b,\pi)\in a$ and thus $c\in\mathrm{Cl}(a)$ and it follows that $\|c\notin\mathcal{V}(a)\|=\Pi$. Therefore, $\eta\Vdash c\notin b$; by hypothesis on ξ, we have $\xi\star\eta\cdot\pi\in\bot\!\!\!\bot$.

(ii) Let $a,\vec{c}$ be individuals in $\mathcal{M}$; we must show $\mathsf{I}\Vdash A\in\mathcal{Q}(a)$, where $A=\{y\in a\ ;\ F[y,\vec{c}]\}$. We have $A=\{(b,\xi\cdot\pi)\ ;\ (b,\pi)\in a,\ \xi\Vdash F[b,\vec{c}]\}$ and therefore $A\subset\mathrm{Cl}(a)\times\Pi$. But we have: $\|A\notin\mathcal{Q}(a)\|=\{\pi\in\Pi\ ;\ (A,\pi)\in\mathcal{Q}(a)\}=\Pi$ and therefore $\mathsf{I}\Vdash A\in\mathcal{Q}(a)$. ◀

We can now define the function symbols $\overline{\bigcup}$ and $\overline{\mathcal{P}}$ by setting:

$$\overline{\bigcup}x = \{z\in\mathcal{V}(x)\ ;\ (\exists y\in x)\, z\in y\} \quad ; \quad \overline{\mathcal{P}}(x) = \{y\in\mathcal{Q}(x)\ ;\ y\subseteq x\}.$$

Collection

We shall use in the following, function symbols associated with a strong form of the *collection scheme*. In order to define these function symbols, it is convenient to decompose them, which is done in Theorems 2, 3 and 4.

▶ **Theorem 2.** *For each formula $F(x,\vec{z})$ of $ZF_\in$, we have:*

$$\Vdash \forall\vec{z}(\exists x\, F(x,\vec{z})\to(\exists x\in\phi_F(\vec{z}))F(x,\vec{z})) \quad ; \quad \Vdash\forall\vec{z}(\forall x\in\phi_F(\vec{z}))F(x,\vec{z})$$

where ϕ_F is a function symbol defined in $\mathcal{M}$.

Proof. We show $\lambda x(x)\mathsf{I}\Vdash\forall x(x\in\Phi_F(\vec{z})\to F(x,\vec{z}))\to\forall x\, F(x,\vec{z})$ where the function symbol Φ_F is defined as follows: By means of the collection scheme in $\mathcal{M}$, we define a function symbol $\Psi(\vec{z})$ such that: $\|\forall x\, F(x,\vec{z})\|=\bigcup_{x\in\Psi(\vec{z})}\|F(x,\vec{z})\|$ and we set $\Phi_F(\vec{z})=\Psi(\vec{z})\times\Pi$. Let $\xi\Vdash\forall x(x\in\Phi_F(\vec{z})\to F(x,\vec{z}))$ and $\pi\in\|\forall x\, F(x,\vec{z})\|$. Then $\pi\in\|F(x,\vec{z})\|$ for some $x\in\Psi(\vec{z})$, and therefore $\mathsf{I}\Vdash x\in\Phi_F(\vec{z})$ and $\xi\star\mathsf{I}\cdot\pi\in\bot\!\!\!\bot$.

Therefore, by replacing F with $\neg F$, we have $\Vdash\exists x\, F(x,\vec{z})\to(\exists x\in\Phi_{\neg F}(\vec{z}))\, F(x,\vec{z})$. Thus, we only need to set $\phi_F(\vec{z})=\{x\in\Phi_{\neg F}(\vec{z})\ ;\ F(x,\vec{z})\}$. ◀

▶ **Theorem 3.** *For every formula $F(y,\vec{z})$ of ZF_ε, we have:*

$$\Vdash \forall\vec{z}\,(\exists x\forall y(F(y,\vec{z})\to y\,\varepsilon\,x)\to\forall y(F(y,\vec{z})\leftrightarrow y\,\varepsilon\,\gamma_F(\vec{z})))$$

where γ_F is a function symbol defined in $\mathcal{M}$.

Proof. By Theorem 2, we have:

$$\Vdash \forall\vec{z}\,(\exists x\forall y(F(y,\vec{z})\to y\,\varepsilon\,x)\to(\exists x\,\varepsilon\,\phi(\vec{z}))\forall y(F(y,\vec{z})\to y\,\varepsilon\,x))$$

where ϕ is a function symbol. Therefore we have, by definition of $\overline{\bigcup}\phi(\vec{z})$:

$$\Vdash \forall\vec{z}\left(\exists x\forall y(F(y,\vec{z})\to y\,\varepsilon\,x)\to\forall y(F(y,\vec{z})\to y\,\varepsilon\,\overline{\bigcup}\phi(\vec{z}))\right).$$

Now, we only need to set $\gamma_F(\vec{z})=\{y\,\varepsilon\,\overline{\bigcup}\phi(\vec{z})\;;\;F(y,\vec{z})\}$ (comprehension scheme). ◀

When the hypothesis $\exists x\forall y(F(y,\vec{z})\to y\,\varepsilon\,x)$ is satisfied, we say that *the formula $F(y,\vec{z})$ defines a set*. For the function symbol $\gamma_F(\vec{z})$, we shall use the more intuitive notation $\{y\;;\;F(y,\vec{z})\}$, where y is a bound variable.

▶ **Theorem 4.** *Let $f(x,\vec{z})$ be a $(n+1)$-ary function symbol (defined in $\mathcal{M}$). Then, we have:*

$$\Vdash \forall a\forall y\forall\vec{z}\,(y\,\varepsilon\,\phi_f(a,\vec{z})\leftrightarrow(\exists x\,\varepsilon\,a)(y=f(x,\vec{z})))$$

where ϕ_f is a $(n+1)$-ary function symbol.

Proof. We define, in $\mathcal{M}$, the symbol ϕ_f as follows: Let $a_0,y_0,\vec{z}_0$ be fixed individuals in $\mathcal{M}$; we set $\phi_f(a_0,\vec{z}_0)=\{(f(x,\vec{z}_0),\pi)\;;\;(x,\pi)\in a_0\}$. Then, we have immediately $\|y_0\notin\phi_f(a_0,\vec{z}_0)\|=\|\forall x(y_0=f(x,\vec{z}_0)\hookrightarrow x\notin a_0)\|$. Therefore: $\Vdash\forall x(y_0=f(x,\vec{z}_0)\hookrightarrow x\notin a_0)\leftrightarrow y_0\notin\phi_f(a_0,\vec{z}_0)$ which gives the desired result. ◀

▶ Remark. The *connective* $\hookrightarrow$ is defined in [7, 9]. It is equivalent to $\to$ but simpler to realize. Its hypothesis must be a strong equality. For the function symbol $\phi_f(a,\vec{z})$, we shall use the more intuitive notation $\{f(x,\vec{z})\;;\;x\,\varepsilon\,a\}$, where x is a bound variable. We call it *image of a by the function $f(x)$*.

Miscellaneous symbols

In the following, we shall use some function symbols, the definition and properties of which are given in [9]. We simply recall their definition below.

- The unary function symbol $\daleth$, defined in $\mathcal{M}$ by $\daleth x=x\times\Pi$. For any individual E of $\mathcal{M}$, the *restricted quantifier* $\forall x^{\daleth E}$ is defined in [7] or [9] by: $\|\forall x^{\daleth E}F[x]\|=\bigcup_{x\in E}\|F[x]\|$ and we have $\Vdash\forall x^{\daleth E}F[x]\leftrightarrow\forall x(x\,\varepsilon\,\daleth E\to F[x])$. In the realizability model $\mathcal{N}$, the formula $x\,\varepsilon\,\daleth E$ may be intuitively understood as *"x is of type E"*. For instance, $\daleth 2$ may be considered as *the type of booleans* and $\daleth\mathbb{N}$ as *the type of integers*.
- The function symbols $\wedge,\vee,\neg$, with domains $\{0,1\}\times\{0,1\}$ and $\{0,1\}$, and values in $\{0,1\}$, are defined in $\mathcal{M}$ by means of the usual truth tables. These functions define, in $\mathcal{N}$, a structure of Boolean algebra on $\daleth 2$. We call it the *characteristic Boolean algebra* of the realizability model $\mathcal{N}$.
- A binary function symbol with domain $\{0,1\}\times\mathcal{M}$, denoted by $(\alpha,x)\mapsto\alpha x$, by setting:

  $$0x=\emptyset\,;\;1x=x\,.$$

  In the model $\mathcal{N}$, the domain of this function is $\daleth 2\times\mathcal{N}$.

- A binary function symbol $\sqcup$ with domain $\mathcal{M}\times\mathcal{M}$, by setting $x \sqcup y = x \cup y$.
  Remark: The extension of this function to the model $\mathcal{N}$ *is not* the union $\cup$, which explains the use of another symbol.

▶ **Lemma 5** (Linearity). *Let f be a binary function symbol, defined in $\mathcal{M}$. Then, we have:*

(i) $\mathsf{I} \Vdash \forall\alpha^{\mathbb{B}2}\forall x\forall y(\alpha f(x,y) = \alpha f(\alpha x, y))$.

(ii) *Moreover, if $f(\emptyset,\emptyset) = \emptyset$, then:*
$\mathsf{I} \Vdash \forall\alpha^{\mathbb{B}2}\forall\alpha'^{\mathbb{B}2}\forall x\forall y\forall x'\forall y'\ (\alpha\wedge\alpha' = 0 \hookrightarrow f(\alpha x \sqcup \alpha' x', \alpha y \sqcup \alpha' y') = \alpha f(x,y) \sqcup \alpha' f(x',y'))$.

Proof. It suffices to check:

- for (i) the two cases $\alpha = 0, 1$;
- for (ii) the three cases $(\alpha,\alpha') = (0,0),(0,1),(1,0)$;

which is is trivial. ◀

Symbols for characteristic functions

Let $R(x_1,\dots,x_n)$ be an n-ary relation defined in $\mathcal{M}$. Its *characteristic function*, with values in $\{0,1\}$, will be denoted by $\langle R(x_1,\dots,x_n)\rangle$. Therefore, we have:

$$\mathcal{M} \models \forall\vec{x}(R(\vec{x}) \leftrightarrow \langle R(\vec{x})\rangle = 1).$$

In the realizability model $\mathcal{N}$, the function symbol $\langle R(\vec{x})\rangle$ takes its values in $\mathbb{B}2$.

The Theorem 8 below shows that, if a binary relation $y \prec x$ is well founded in $\mathcal{M}$, then the relation $\langle y \prec x\rangle = 1$ is well founded in $\mathcal{N}$.

4 Well founded relations

In this section, we study properties of well founded relations in $\mathcal{N}$. All the results obtained here are, of course, trivial in ZF. The difficulties come from the fact that the relation ε of strong membership does not satisfy extensionality.

Given a binary relation $\prec$, an individual a is said *minimal for* $\prec$ if we have $\forall x\,\neg(x \prec a)$. The binary relation $\prec$ is called *well founded* if we have:

$$\forall X\ (\forall x(\forall y(y \prec x \to y \notin X) \to x \notin X) \to \forall x(x \notin X))\ .$$

The intuitive meaning is that each non empty individual X has an ε-element minimal for $\prec$. Theorem 6 shows that this also true for non empty classes.

▶ **Theorem 6.** *If the relation $x \prec y$ is well founded then, for every formula $F[x,\vec{z}]$ of ZF_ε, we have:*

$$\forall\vec{z}\,(\forall x(\forall y(y \prec x \to F[y,\vec{z}]) \to F[x,\vec{z}]) \to \forall x\,F[x,\vec{z}])\ .$$

Proof. By contradiction; we consider, in $\mathcal{N}$, an individual a and a formula $G[x]$ such that:

(1) $G[a]\ ;\ \forall x\,(G[x] \to \exists y\{G[y], y \prec x\})$.

We apply the axiom scheme of infinity of ZF_ε:

(2) $\exists b\,\{a\,\varepsilon\,b, (\forall x\,\varepsilon\,b)\,(\exists y\,H(x,y) \to (\exists y\,\varepsilon\,b)H(x,y))\}$ by setting $H(x,y) \equiv G[x]\wedge G[y]\wedge y \prec x$.
Let $X = \{x\,\varepsilon\,b\ ;\ G(x)\}$.

By (1) and (2), we get $a \varepsilon X$.

We obtain a contradiction with the hypothesis, by showing $(\forall x \varepsilon X)(\exists y \varepsilon X)(y \prec x)$: suppose $x \varepsilon b$ and $G[x]$; by (2), we have:

$$\exists y\{G[x], G[y], y \prec x\} \to (\exists y \varepsilon b)\{G[x], G[y], y \prec x\}\,.$$

By $G[x]$ and (1), we have $\exists y\{G[x], G[y], y \prec x\}$. Therefore, we have $(\exists y \varepsilon b)\{G[y], y \prec x\}$, hence the result. ◀

Therefore, in order to show $\forall x\, F[x]$, it suffices to show $\forall x\, (\forall y(y \prec x \to F[y]) \to F[x])$. Then, we say that we have shown $\forall x\, F[x]$ *by induction on* x*, following the well founded relation* $\prec$.

▶ **Theorem 7.** *The binary relation* $x \in y$ *is well founded.*

Proof. We must show $\forall x(\forall y(y \in x \to y \notin X) \to x \notin X) \to \forall x(x \notin X)$. We apply Theorem 6 to the well founded relation $x \varepsilon y$ and the formula $F[x] \equiv x \notin X$. This gives: $\forall x(\forall y(y \varepsilon x \to y \notin X) \to x \notin X) \to \forall x(x \notin X)$. Now, we have immediately $\Vdash x \notin X \to x \notin X$. Thus, it remains to show: $\Vdash \forall x(\forall y(y \in x \to y \notin X) \to x \notin X) \to \forall x(\forall y(y \varepsilon x \to y \notin X) \to x \notin X)$. But we have $x \notin X \equiv \forall x'(x' \simeq x \to x' \notin X)$. Therefore, we need to show: $\Vdash \forall x(\forall y(y \in x \to y \notin X) \to x \notin X), \forall y(y \varepsilon x \to y \notin X), x' \simeq x \to x' \notin X$. It is enough to show: $\Vdash \forall y(y \varepsilon x \to y \notin X), x' \simeq x \to \forall y(y \in x' \to y \notin X)$. Now, from $x' \simeq x, y \in x'$, we deduce $y \in x$. Thus, there is some $y' \simeq y$ such that $y' \varepsilon x$. Then, from $\forall y(y \varepsilon x \to y \notin X)$, we deduce $y' \notin X$, and therefore $y \notin X$. ◀

For instance, in the following, we shall use the fact that, if there is an ordinal ρ such that $F[\rho]$, then there exists a least such ordinal, for any formula $F[\rho]$ *written in the language of* ZF_ε. This follows from Theorem 7.

Preservation of well-foundedness

▶ **Theorem 8.** *Let* $\prec$ *be a well founded binary relation defined in the ground model* $\mathcal{M}$*. Then, the relation* $\langle y \prec x\rangle = 1$ *is well founded in* $\mathcal{N}$*. In fact, we have:*

$$\mathsf{Y} \Vdash \forall X\, (\forall x(\forall y(\langle y \prec x\rangle = 1 \hookrightarrow y \notin X) \to x \notin X) \to \forall x(x \notin X))$$

where $\mathsf{Y} = (\lambda x \lambda f(f)(x)xf)\lambda x \lambda f(f)(x)xf$ *(Turing fixpoint combinator).*

Proof. Let $\xi \in \Lambda$ be such that $\xi \Vdash \forall x(\forall y(\langle y \prec x\rangle = 1 \hookrightarrow y \notin X_0) \to x \notin X_0)$, X_0 being any individual in $\mathcal{M}$. We set $F[x] \equiv (\forall \pi \in \|x \notin X_0\|)(\mathsf{Y} \star \xi \cdot \pi \in \perp\!\!\!\perp)$, and we have to show $\forall x\, F[x]$. Since $\prec$ is a well founded relation, it suffices to show $\forall x\, (\forall y(y \prec x \to F[y]) \to F[x])$, or equivalently $\neg F[x_0] \to (\exists y \prec x_0)\neg F[y]$, for any individual x_0. By the hypothesis $\neg F[x_0]$, there exists $\pi_0 \in \|x_0 \notin X_0\|$ such that $\mathsf{Y} \star \xi \cdot \pi_0 \notin \perp\!\!\!\perp$ and therefore, we have $\xi \star \mathsf{Y}\xi \cdot \pi_0 \notin \perp\!\!\!\perp$. By hypothesis on ξ, we deduce $\mathsf{Y}\xi \not\Vdash \forall y(\langle y \prec x_0\rangle = 1 \hookrightarrow y \notin X_0)$. Thus, there exists $y_0 \prec x_0$ such that $\mathsf{Y}\xi \not\Vdash y_0 \notin X_0$. Therefore, we have $(\exists \pi \in \|y_0 \notin X_0\|)(\mathsf{Y} \star \xi \cdot \pi \notin \perp\!\!\!\perp)$, that is $\neg F[y_0]$. ◀

Definition of a rank function

Definition. A *function with domain* D is an individual ϕ such that: $(\forall z \varepsilon \phi)(\exists x \varepsilon D)\exists y(z = (x, y))$; $(\forall x \varepsilon D)\exists y((x, y) \varepsilon \phi)$; $\forall x \forall y \forall y'((x, y) \varepsilon \phi, (x, y') \varepsilon \phi \to y = y')$.

Let ϕ be a function with domain D and $F[y, \vec{z}]$ a formula of ZF_ε. Then, the formula: $\exists y\{(x, y) \varepsilon \phi,\ F[y, \vec{z}]\}$ is denoted by $F[\phi(x), \vec{z}]$.

▶ Remark. Beware, despite the same notation $\phi(x)$, it is not a function symbol.

By means of Theorem 3, we define the binary function symbol Im by setting:

$$\mathrm{Im}(\phi, D) = \{y\ ;\ (\exists x\,\varepsilon\, D)\,(x,y)\,\varepsilon\,\phi\}\,.$$

When ϕ is a function with domain D, we shall use, for $\mathrm{Im}(\phi, D)$, the more intuitive notation $\{\phi(x)\ ;\ x\,\varepsilon\, D\}$, which we call *image of the function* ϕ.

Let $D' \subseteq D$, that is $\forall x(x \notin D \to x \notin D')$; *a restriction of* ϕ *to* D' is, by definition, a function ϕ' with domain D' such that $\phi' \subseteq \phi$. For instance, $\{z\,\varepsilon\,\phi\ ;\ (\exists x\,\varepsilon\, D')\exists y(z = (x,y))\}$ is a restriction of ϕ to D'. If ϕ'_0, ϕ'_1 are both restrictions of ϕ to D', then $\phi'_0 \cong \phi'_1$.

Definition. A binary relation $\prec$ is called *ranked*, if we have $\forall x\exists y\forall z(z \prec x \to z\,\varepsilon\, y)$, in other words: the minorants of any individual form a set. By Theorem 3, if the relation $\prec$ is ranked and defined by a formula $P[x, y, \vec{u}]$ of ZF_ε with parameters $\vec{u}$ in $\mathcal{N}$, we have: $\mathcal{N} \models \forall x\forall y(x \prec y \leftrightarrow x\,\varepsilon\, f(y, \vec{u}))$, *for some symbol of function* f*, defined in* $\mathcal{M}$.

In what follows, we suppose that $\prec$ is a ranked *transitive* binary relation.

A function ϕ with domain $\{x\ ;\ x \prec a\}$ will be called *a-inductive for* $\prec$, if we have: $\phi(x) \simeq \{\phi(y)\ ;\ y \prec x\}$ for every $x \prec a$. In other words: $(\forall x \prec a)(\forall y \prec x)\,\phi(y) \in \phi(x)$; $(\forall x \prec a)(\forall z\,\varepsilon\,\phi(x))(\exists y \prec x)\,z \simeq \phi(y)$.

If ϕ is a-inductive for $\prec$, we set $\mathsf{O}(\phi, a) = \{\phi(x)\ ;\ x \prec a\}$ (image of ϕ).

▶ **Lemma 9.** *Let* ϕ, ϕ' *be two functions,* a*-inductive for* $\prec$*. Then:*

(i) $\phi(x) \simeq \phi'(x)$ *for every* $x \prec a$.

(ii) $\mathsf{O}(\phi, a) \simeq \mathsf{O}(\phi', a)$.

(iii) $(\forall x \prec a)\mathrm{On}(\phi(x))$*;* $\mathsf{O}(\phi, a)$ *is an ordinal, called* ordinal of ϕ.

Proof.

(i) By induction on $\phi(x)$, following $\in$: if $u\,\varepsilon\,\phi(x)$, then $u \simeq \phi(y)$ with $y \prec x$. Since $\phi(y) \in \phi(x)$, we have $\phi(y) \simeq \phi'(y)$ by the induction hypothesis; therefore $\phi(y) \in \phi'(x)$ and $\phi(x) \subset \phi'(x)$. Conversely, if $u\,\varepsilon\,\phi'(x)$, then $u \simeq \phi'(y)$ with $y \prec x$. Thus, we have $\phi(y) \in \phi(x)$, and therefore $\phi(y) \simeq \phi'(y)$ by the induction hypothesis; therefore $u \in \phi(x)$ and $\phi'(x) \subset \phi(x)$.

(ii) Immediate, by (i).

(iii) We show $\mathrm{On}(\phi(x))$ by induction on $\phi(x)$, for the well founded relation $\in$: If $u\,\varepsilon\,\phi(x)$, we have $u \simeq \phi(y)$ with $y \prec x$; therefore, we have $\mathrm{On}(u)$ by the induction hypothesis. If $v\,\varepsilon\,u$, then $v\,\varepsilon\,\phi(y)$, therefore $v \simeq \phi(z)$ with $z \prec y$; therefore $v \in \phi(x)$. It follows that $\phi(x)$ is a transitive set of ordinals, thus an ordinal. Then, $\mathsf{O}(\phi, a)$ is also a transitive set of ordinals, and therefore an ordinal. ◀

▶ **Lemma 10.** *If* ϕ *is* a*-inductive for* $\prec$*, and if* $b \prec a$*, then every restriction* ψ *of* ϕ *to the domain* $\{x\ ;\ x \prec b\}$ *is a* b*-inductive function for* $\prec$*.*

Proof. Indeed, we have, $\psi(x) = \phi(x) \simeq \{\phi(y)\ ;\ y \prec x\} \simeq \{\psi(y)\ ;\ y \prec x\}$. ◀

By means of Theorem 2, we define a unary function symbol Φ, such that:

- $\forall x(\forall f\,\varepsilon\,\Phi(x))(f$ is a x-inductive function$)$;
- $\forall x\forall f\Big(f$ is a x-inductive function $\to \exists f(f\,\varepsilon\,\Phi(x))\Big)$.

In other words, $\Phi(x)$ is a set of x-inductive functions, which is non void if there exists at least one such function. Finally, we define the unary function symbol Rk, using Theorem 4, by setting:

$$\mathrm{Rk}(x) = \overline{\bigcup}\{\mathsf{O}(f,x)\ ;\ f\,\varepsilon\,\Phi(x)\}$$

(the symbol $\overline{\bigcup}$ is defined after Theorem 1). Therefore, $\mathrm{Rk}(x)$ is the union of the ordinals of the x-inductive functions in the set $\Phi(x)$. Since all these ordinals are extensionally equivalent, by Lemma 9(ii), their union $\mathrm{Rk}(x)$ is also an equivalent ordinal.

▶ Remark. If there exists no x-inductive function, then $\mathrm{Rk}(x)$ is void. The function symbols $\mathsf{O}, \Phi, \mathrm{Rk}$ have additional arguments, which are the parameters $\vec{u}$ of the formula $P[x,y,\vec{u}]$ which defines the relation $y \prec x$.

We suppose now that $\prec$ is a ranked transitive relation, which is *well founded*. It is therefore a *strict ordering*.

▶ **Lemma 11.** *Every restriction of* Rk *to the domain* $\{x\ ;\ x \prec a\}$ *is an* a*-inductive function for* $\prec$*.*

Proof. By induction on a, following $\prec$.

Let f be a restriction of Rk to the domain $\{x\ ;\ x \prec a\}$ and let $x \prec a$. We must show that $f(x) \simeq \{f(y)\ ;\ y \prec x\}$, in other words, that we have:

$$\mathrm{Rk}(x) \simeq \{\mathrm{Rk}(y)\ ;\ y \prec x\}\,.$$

Let ψ be any restriction of Rk to the domain $\{y\ ;\ y \prec x\}$. By the induction hypothesis, ψ is a x-inductive function for $\prec$. We now show that $\mathrm{Rk}(x) \simeq \{\mathrm{Rk}(y)\ ;\ y \prec x\}$:

(i) If $u\,\varepsilon\,\mathrm{Rk}(x)$, then $u\,\varepsilon\,\mathsf{O}(\phi,x)$ for some function ϕ which is x-inductive for $\prec$, *provided that there exists such a function*. Now, there exists effectively one, otherwise $\mathrm{Rk}(x)$ would be void. Therefore, by definition of $\mathsf{O}(\phi,x)$, we have $u = \phi(y)$ with $y \prec x$. But $\mathrm{Rk}(y) \simeq \phi(y)$, since ϕ, ψ are both x-inductive functions for $\prec$, and $\psi(y) = \mathrm{Rk}(y)$ (Lemma 9(i)). Therefore, we have $u \simeq \mathrm{Rk}(y)$, with $y \prec x$.

(ii) Conversely, if $y \prec x$, then $\mathrm{Rk}(y) = \psi(y)$. Let $\phi\,\varepsilon\,\Phi(x)$; then ϕ, ψ are x-inductive for $\prec$; therefore $\phi(y) \simeq \psi(y)$ (Lemma 9(i)). Now $\phi(y)\,\varepsilon\,\mathsf{O}(\phi,x)$, and therefore $\phi(y)\,\varepsilon\,\mathrm{Rk}(x)$ by definition of $\mathrm{Rk}(x)$. It follows that $\mathrm{Rk}(y) = \psi(y) \in \mathrm{Rk}(x)$. ◀

▶ **Theorem 12.** *We have* $\mathrm{Rk}(x) \simeq \{\mathrm{Rk}(y)\ ;\ y \prec x\}$ *for every* x*.*

Proof. By induction on x, following $\prec$; let ψ be any restriction of Rk to the domain $\{y\ ;\ y \prec x\}$. By Lemma 11, ψ is a x-inductive function for $\prec$. Then, we finish the proof, by repeating paragraphs (i) and (ii) of the proof of Lemma 11. ◀

Rk is called the *rank function* of the ranked, well founded and transitive relation $\prec$. $\mathrm{Rk}(x)$ is, for every x, a representative of the ordinal of any x-inductive function for $\prec$.

The values of the rank function Rk form an initial segment of On, which we shall call *the image of* Rk. It is therefore, *either an ordinal, or the whole of* On.

▶ **Lemma 13.** *Let* $\prec_0, \prec_1$ *be two ranked transitive well founded relations, and* f *a function such that* $\forall x \forall y (x \prec_0 y \to f(x) \prec_1 f(y))$*. If* $\mathrm{Rk}_0, \mathrm{Rk}_1$ *are their rank functions, then we have* $\forall x\,(\mathrm{Rk}_0(x) \leq \mathrm{Rk}_1(f(x)))$*, and the image of* Rk_0 *is an initial segment of the image of* Rk_1*.*

Proof. We show immediately $\forall x\,(\mathrm{Rk}_0(x) \leq \mathrm{Rk}_1(f(x)))$ by induction following $\prec_0$. Hence the result, since the image of a rank function is an initial segment of On. ◀

5 An ultrafilter on $\rightthreetimes 2$

In all of the following, we write $y < x$ for $y \in \mathrm{Cl}(x)$ in $\mathcal{M}$, where $\mathrm{Cl}(x)$ denotes the transitive closure of x. It is a strict well founded ordering (many other such orderings would do the job, for instance the relation $\mathrm{rank}(y) < \mathrm{rank}(x)$). The binary function symbol $\langle y < x\rangle$ is therefore defined in $\mathcal{N}$, with values in $\rightthreetimes 2$. By Theorem 8, the binary relation $\langle y < x\rangle = 1$ is well founded in $\mathcal{N}$.

▶ **Theorem 14.** $\Vdash$ *There exists an ultrafilter $\mathcal{D}$ on $\rightthreetimes 2$, which is defined as follows: $\mathcal{D} = \{\alpha \,\varepsilon\, \rightthreetimes 2\ ;$ the relation $\langle y < x\rangle \geq \alpha$ is well founded $\}$.*

The formula $\alpha \,\varepsilon\, \mathcal{D}$, which we shall also write $\mathcal{D}[\alpha]$, is therefore:

$$\mathcal{D}[\alpha] \equiv \forall X\,(\forall x(\forall y(\langle y < x\rangle \geq \alpha \hookrightarrow y \notin X) \to x \notin X) \to \forall x(x \notin X))\ .$$

▶ Remark. By Lemma 5, the formula $\langle y < x\rangle \geq \alpha$ may be written $\langle \alpha y < \alpha x\rangle = \alpha$. We have:

- $\mathcal{D}[1] \equiv \forall X\,(\forall x(\forall y(\langle y < x\rangle = 1 \hookrightarrow y \notin X) \to x \notin X) \to \forall x(x \notin X))$.
- $\mathcal{D}[0] \equiv \forall X((\emptyset \notin X \to \emptyset \notin X) \to \emptyset \notin X)$.

Proof. We have immediately: $\lambda x\, x \Vdash \neg\mathcal{D}[0]$; $\mathsf{Y} \Vdash \mathcal{D}[1]$; $\mathsf{I} \Vdash \forall\alpha^{\rightthreetimes 2}\forall\beta^{\rightthreetimes 2}\,(\alpha \leq \beta \hookrightarrow (\mathcal{D}[\alpha] \to \mathcal{D}[\beta]))$ (more precisely: $\|\mathcal{D}[1]\| \subset \|\mathcal{D}[0]\|$).

Therefore, in order to prove Theorem 14, it suffices to show:

- $\Vdash \forall\alpha^{\rightthreetimes 2}\forall\beta^{\rightthreetimes 2}\,(\alpha\wedge\beta = 0 \hookrightarrow (\mathcal{D}[\alpha\vee\beta] \to \mathcal{D}[\alpha] \vee \mathcal{D}[\beta]))$; see Theorem 15;
- $\Vdash \forall\alpha^{\rightthreetimes 2}\forall\beta^{\rightthreetimes 2}\,(\alpha\wedge\beta = 0 \hookrightarrow (\mathcal{D}[\alpha], \mathcal{D}[\beta] \to \bot))$; or even only:
- $\Vdash \forall\alpha^{\rightthreetimes 2}(\mathcal{D}[\alpha], \mathcal{D}[\neg\alpha] \to \bot)$; see Theorem 22. ◀

Notation. For $\alpha \,\varepsilon\, \rightthreetimes 2$, we shall write $x <_\alpha y$ for $\langle x < y\rangle \geq \alpha$.

▶ **Theorem 15.**

(i) $\Vdash \forall\alpha^{\rightthreetimes 2}\forall\beta^{\rightthreetimes 2}\,(\alpha\wedge\beta = 0 \hookrightarrow (\mathcal{D}[\alpha\vee\beta] \to \mathcal{D}[\alpha] \vee \mathcal{D}[\beta]))$.

(ii) $\Vdash \forall\alpha^{\rightthreetimes 2}\forall\beta^{\rightthreetimes 2}\,(\mathcal{D}[\alpha\vee\beta] \to \mathcal{D}[\alpha] \vee \mathcal{D}[\beta])$.

Proof.

(i) Let $\alpha, \beta \,\varepsilon\, \rightthreetimes 2$ be such that $\alpha\wedge\beta = 0, \neg\mathcal{D}[\alpha], \neg\mathcal{D}[\beta]$. We have to show $\neg\mathcal{D}[\alpha\vee\beta]$. By hypothesis on α and β, there exists individuals a_0, A (resp. b_0, B) such that $a_0 \,\varepsilon\, A$ (resp. $b_0 \,\varepsilon\, B$) and A (resp. B) has no minimal ε-element for $<_\alpha$ (resp. for $<_\beta$). We set:

$$c_0 = \alpha a_0 \sqcup \beta b_0 \quad \text{and} \quad C = \{\alpha x \sqcup \beta y\ ;\ x \,\varepsilon\, A, y \,\varepsilon\, B\}\ .$$

Therefore, we have $c_0 \,\varepsilon\, C$; it suffices to show that C has no minimal ε-element for $<_{\alpha\vee\beta}$. Let $c \,\varepsilon\, C, c = \alpha a \sqcup \beta b$, with $a \,\varepsilon\, A, b \,\varepsilon\, B$. By hypothesis on A, B, there exists $a' \,\varepsilon\, A$ and $b' \,\varepsilon\, B$ such that $a' <_\alpha a$, $b' <_\beta b$. If we set $c' = \alpha a' \sqcup \beta b'$, we have $c' \,\varepsilon\, C$, as needed. We also have: $\langle c' = a'\rangle \geq \alpha, \langle a' < a\rangle \geq \alpha, \langle c = a\rangle \geq \alpha$; it follows that $\langle c' < c\rangle \geq \alpha$. In the same way, we have $\langle c' < c\rangle \geq \beta$ and therefore, finally, $\langle c' < c\rangle \geq \alpha\vee\beta$.

(ii) We set $\beta' = \beta\wedge(\neg\alpha)$; we have $\alpha\wedge\beta' = 0$ and $\alpha\vee\beta' = \alpha\vee\beta$. Therefore, we have: $\mathcal{D}[\alpha\vee\beta] \to \mathcal{D}[\alpha] \vee \mathcal{D}[\beta']$. Now, we have $\beta' \leq \beta$ and therefore $\mathcal{D}[\beta'] \to \mathcal{D}[\beta]$. ◀

▶ **Lemma 16.**

(i) $\mathsf{I} \Vdash \forall x\forall y(\langle x < y\rangle \neq 1 \to x \notin y)$.

(ii) *If $\mathcal{M} \models u \in v$, then* $\mathsf{I} \Vdash u \,\varepsilon\, \rightthreetimes v$.

(iii) $\mathsf{I} \Vdash \forall x\forall y\forall\alpha^{\rightthreetimes 2}\,(\langle x < y\rangle \geq \alpha \hookrightarrow \alpha x \,\varepsilon\, \rightthreetimes\mathrm{Cl}(\{y\}))$.

(iv) $\Vdash \forall x\forall y\,(\langle x < y\rangle = 1 \leftrightarrow x \,\varepsilon\, \rightthreetimes\mathrm{Cl}(y))$.

Proof.

(i) Let a, b be two individuals. Let $\xi \Vdash \langle a < b\rangle \neq 1$, $\pi \in \|a \notin b\|$; then $(a,\pi) \in b$ and therefore $\langle a < b\rangle = 1$ and $\xi \Vdash \bot$; thus $\xi \star \pi \in \perp\!\!\!\perp$.

(ii) Indeed, we have $\|u \notin \daleth v\| = \{\pi \in \Pi\ ;\ (u,\pi) \in v \times \Pi\} = \Pi$.

(iii) Let $\alpha \in \{0,1\}$ and $a, b \in \mathcal{M}$ such that $\langle a < b\rangle \geq \alpha$. If $\alpha = 0$, we must show $\mathsf{I} \Vdash \emptyset\, \varepsilon\, \daleth \mathrm{Cl}(\{y\})$ which follows from (ii). If $\alpha = 1$, then $\langle a < b\rangle = 1$, that is $a \in \mathrm{Cl}(b)$, therefore $a \in \mathrm{Cl}(\{b\})$. From (ii), it follows that $\mathsf{I} \Vdash a\, \varepsilon\, \daleth \mathrm{Cl}(\{b\})$.

(iv) Indeed, if a, b are individuals of $\mathcal{M}$, we have trivially: $\|\langle a < b\rangle \neq 1\| = \|a \notin \daleth \mathrm{Cl}(b)\|$. ◀

▶ **Lemma 17.** *The well founded relation $\langle x < y\rangle = 1$ is ranked, and its rank function R has for image the whole of On.*

Proof. Lemma 16(iv) shows that this relation is ranked. Let ρ be an ordinal and r an individual $\simeq \rho$. We show, by induction on ρ, that $\mathsf{R}(r) \geq \rho$. Indeed, for every $\rho' \in \rho$, there exists $r'\, \varepsilon\, r$ such that $r' \simeq \rho'$. We have $\mathsf{R}(r') \geq \rho'$ by induction hypothesis, and $\langle r' < r\rangle = 1$ from Lemma 16(i). Therefore, we have $\rho' \in \mathsf{R}(r)$ by definition of R, and finally $\mathsf{R}(r) \geq \rho$. This shows that the image of R is not bounded in On. Since it is an initial segment, it is the whole of On. ◀

▶ **Theorem 18.** *Let $F(x,y)$ be a formula of ZF_ε, with parameters. Then, we have:*

$$\mathsf{I} \Vdash \forall x \forall y \left(\forall \varpi^{\daleth\Pi} F(x, f(x,\varpi)) \to F(x,y)\right)$$

for some function symbol f, defined dans $\mathcal{M}$, with domain $\mathcal{M} \times \Pi$.

Proof. Since the ground model $\mathcal{M}$ satisfies V = L (or only the *choice principle*), we can define, in $\mathcal{M}$, a function symbol f such that:

$$\forall x \forall y (\forall \varpi \in \Pi)\, (\varpi \in \|F(x,y)\| \to \varpi \in \|F(x, f(x,\varpi))\|)\ .$$

Let a, b be individuals, $\xi \Vdash \forall \varpi^{\daleth\Pi} F(a, f(a,\varpi))$ and $\pi \in \|F(a,b)\|$. Thus, we have $\pi \in \|F(a, f(a,\pi))\|$, and therefore $\xi \star \pi \in \perp\!\!\!\perp$. ◀

Definitions. Let a be any individual of $\mathcal{N}$ and κ an ordinal (therefore, κ is not an individual of $\mathcal{N}$, but an equivalence class for $\simeq$). A *function* or *application* from κ into a is, by definition, a binary relation $R(\rho, x)$ such that: $\forall x \forall x' (\forall \rho, \rho' \in \kappa)\, (R(\rho,x), R(\rho',x'), \rho \simeq \rho' \to x = x'))$; $(\forall \rho \in \kappa)(\exists x\, \varepsilon\, a) R(\rho, x)$. It is an *injection* if we have $\forall x (\forall \rho, \rho' \in \kappa)\, (R(\rho,x), R(\rho',x) \to \rho \simeq \rho')$. A *surjection* from a onto κ is a function f of domain a such that: $(\forall \rho \in \kappa)(\exists x\, \varepsilon\, a)\, f(x) \simeq \rho$.

▶ **Theorem 19.** *For any individual a, there exists an ordinal κ, such that there is no surjection from a onto κ.*

Proof. Let f be a surjection from a onto an ordinal ρ. We define a strict ordering relation $\prec_f$ by setting $x \prec_f y \Leftrightarrow x\, \varepsilon\, a \wedge y\, \varepsilon\, a \wedge f(x) < f(y)$. It is clear that this relation is well founded, that f is an a-inductive function, and that $\mathsf{O}(f,a) \simeq \rho$. We may consider this relation as a subset of $a \times a$. By means of the axioms of union, power set and collection given above (Theorems 1 to 4), we define an ordinal κ_0, which is the union of the $\mathsf{O}(f,a)$ for all the functions f which are a-inductive for some well founded strict ordering relation on a. In fact, we consider the set:

$$\mathcal{B}(a) = \{X\, \varepsilon\, \overline{\mathcal{P}}(a \times a)\ ;\ X \text{ is a well founded strict ordering relation on } a\}\ .$$

Then, we set $\kappa_0 = \bigcup\{\mathsf{O}(f,a)\ ;\ X\, \varepsilon\, \mathcal{B}(a), f\, \varepsilon\, \Phi(X,a)\}$. In this definition, we use the function symbol Φ, defined after Lemma 10, which associates with each well founded strict ordering relation X on a, a *non void* set of a-inductive functions for this relation.

Then, there exists no surjection from a onto $\kappa_0 + 1$. ◀

Notations. We denote by Δ the first ordinal of $\mathcal{N}$ such that there is no surjection from $\beth\Pi$ onto Δ: for every function ϕ, there exists $\delta \in \Delta$ such that $\forall x^{\beth\Pi}(\phi(x) \not\simeq \delta)$. For each $\alpha \varepsilon \beth 2$, we denote by $\mathcal{N}_\alpha$ the class defined by the formula $x = \alpha x$.

▶ **Lemma 20.** *Let $\alpha_0, \alpha_1 \varepsilon \beth 2$, $\alpha_0 \wedge \alpha_1 = 0$ and R_0 (resp. R_1) be a functional relation of domain $\mathcal{N}_{\alpha_0}$ (resp. $\mathcal{N}_{\alpha_1}$) with values in* On*. Then, either R_0, or R_1, is not surjective onto Δ.*

Proof. By contradiction: we suppose that R_0 and R_1 are both surjective onto Δ. We apply Theorem 18 to the formula $F(x_0, x_1) \equiv \neg(R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 x_1))$, and we get:

$$\forall x_0 \left(\exists x_1 (R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 x_1)) \to \exists \varpi^{\beth\Pi}(R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 f(x_0, \varpi)))\right)$$

where f is a suitable function symbol (therefore *defined in* $\mathcal{M}$). Replacing x_0 with $\alpha_0 x_0$, we obtain:

$$\forall x_0 \left(\exists x_1 (R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 x_1)) \to \exists \varpi^{\beth\Pi}(R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 f(\alpha_0 x_0, \varpi)))\right) .$$

But, by Lemma 5(i), we have $\alpha_1 f(\alpha_0 x, \varpi) = \alpha_1 f(\alpha_1 \alpha_0 x, \varpi) = \alpha_1 f(\emptyset, \varpi)$. It follows that:

$$\forall x_0 \left(\exists x_1 (R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 x_1)) \to \exists \varpi^{\beth\Pi}(R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 f(\emptyset, \varpi)))\right) .$$

By hypothesis, we have $(\forall \rho \in \Delta)\exists x_0 \exists x_1 (\rho \simeq R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 x_1))$.

It follows that: $(\forall \rho \in \Delta)\exists x_0 \exists \varpi^{\beth\Pi}(\rho \simeq R_0(\alpha_0 x_0) \simeq R_1(\alpha_1 f(\emptyset, \varpi)))$; therefore, we have: $(\forall \rho \in \Delta)\exists \varpi^{\beth\Pi}(\rho \simeq R_1(\alpha_1 f(\emptyset, \varpi)))$.

Therefore, the function $\varpi \mapsto R_1(\alpha_1 f(\emptyset, \varpi))$ is a surjection from $\beth\Pi$ onto Δ. But this is a contradiction with the definition de Δ.

Remark. We should write $f(\alpha_0, \alpha_1, x_0, \varpi)$ instead of $f(x_0, \varpi)$, since the function symbol f depends on the four variables $\alpha_0, \alpha_1, x_0, \varpi$. In fact, it depends also on the parameters which appear in R_0, R_1. The proof does not change. ◀

▶ **Corollary 21.** *Let $\alpha_0, \alpha_1 \varepsilon \beth 2$, $\alpha_0 \wedge \alpha_1 = 0$, and $\prec_0, \prec_1$ be two well founded ranked strict ordering relations with respective domains $\mathcal{N}_{\alpha_0}, \mathcal{N}_{\alpha_1}$. Let* Rk_0, Rk_1 *be their rank functions. Then, either the image of* Rk_0*, or that of* Rk_1 *is an ordinal $< \Delta$.*

Proof. In order to be able to define the rank functions Rk_0, Rk_1, we consider the relations $\prec'_0, \prec'_1$, with domain the whole of $\mathcal{N}$, defined by $x \prec'_i y \equiv (x = \alpha_i x) \wedge (y = \alpha_i y) \wedge (x \prec_i y)$ for $i = 0, 1$. These strict ordering relations are well founded and ranked. Their rank functions Rk'_0, Rk'_1 take the value 0 outside $\mathcal{N}_{\alpha_0}, \mathcal{N}_{\alpha_1}$ respectively: indeed, all the individuals outside $\mathcal{N}_{\alpha_i}$ are minimal for $\prec'_i$.

By Lemma 20, one of them, Rk'_0 for instance, is not surjective onto Δ. Since the image of any rank function is an initial segment of On, the image of Rk_0 is an ordinal $< \Delta$. ◀

▶ **Theorem 22.**

(i) $\Vdash \forall \alpha_0^{\beth 2} \forall \alpha_1^{\beth 2} (\alpha_0 \wedge \alpha_1 = 0 \hookrightarrow (\mathcal{D}[\alpha_0], \mathcal{D}[\alpha_1] \to \bot))$.

(ii) $\Vdash \forall \alpha_0^{\beth 2} \forall \alpha_1^{\beth 2} (\mathcal{D}[\alpha_0], \mathcal{D}[\alpha_1] \to \mathcal{D}[\alpha_0 \wedge \alpha_1])$.

Proof.

(i) In $\mathcal{N}$, let $\alpha_0, \alpha_1 \varepsilon \beth 2$ be such that $\alpha_0 \wedge \alpha_1 = 0$ and the relations $\langle x < y \rangle \geq \alpha_0$, $\langle x < y \rangle \geq \alpha_1$ be well founded. Therefore, we have $\alpha_0, \alpha_1 \neq 0$ (and thus, $\alpha_0, \alpha_1 \neq 1$). Therefore, the relations $x \prec_i y \equiv (x = \alpha_i x) \wedge (y = \alpha_i y) \wedge (\langle x < y \rangle = \alpha_i)$ for $i = 0, 1$, are well founded strict orderings. From Lemma 16(iii), it follows that these relations are *ranked*. Now, by Lemma 5, we have: $\Vdash \forall x \forall y \forall \alpha^{\beth 2} (\langle x < y \rangle = 1 \to \langle \alpha x < \alpha y \rangle = \alpha)$. But, by Lemma 17,

the rank function of the well founded relation $\langle x < y\rangle = 1$ has for image the whole of On. Therefore, by Lemma 13, the same is true for the rank functions of the well founded strict order relations $x \prec_0 y$ and $x \prec_1 y$. But this contradicts Corollary 21.

(ii) We have $\alpha_0 \leq (\alpha_0 \wedge \alpha_1) \vee (\neg\alpha_1)$. Therefore, by $\mathcal{D}[\alpha_0]$ and Theorem 15, we have $\mathcal{D}[\alpha_0 \wedge \alpha_1]$ or $\mathcal{D}[\neg\alpha_1]$. But $\mathcal{D}[\neg\alpha_1]$ is impossible, by $\mathcal{D}[\alpha_1]$ and (i). ◀

▶ **Corollary 23.** *$\mathcal{D}[\alpha]$ is equivalent with each one of the following propositions:*

(i) *There exists a well founded ranked strict ordering relation $\prec$ with domain $\mathcal{N}_\alpha$, the rank function of which has an image $\geq \Delta$.*

(ii) *There exists a function with domain $\mathcal{N}_\alpha$ which is surjective onto Δ.*

Proof.

$\mathcal{D}[\alpha] \Rightarrow$ **(i):** By definition of $\mathcal{D}[\alpha]$, the binary relation $(x = \alpha x) \wedge (y = \alpha y) \wedge (\langle x < y\rangle = \alpha)$ is well founded. By Lemma 16(iii), this relation is ranked. We have seen, in the proof of Theorem 22, that the image of its rank function is the whole of On.

(i) $\Rightarrow$ **(ii):** obvious.

(ii) $\Rightarrow \mathcal{D}[\alpha]$**:** Since $\mathcal{D}$ is an ultrafilter, to show $\neg\mathcal{D}[\neg\alpha]$. But, (ii) and $\mathcal{D}[\neg\alpha]$ contradict Lemma 20. ◀

▶ **Theorem 24.** *If $\daleth 2$ is non trivial, there exists no set, which is totally ordered by ε, the ordinal of which is $\geq \Delta$.*

Proof. Let $\alpha \,\varepsilon\, \daleth 2, \alpha \neq 0, 1$ and X be a set which is totally ordered by ε, and equipotent with Δ. Then, we show that the application $x \mapsto \alpha x$ is an injection from X into $\mathcal{N}_\alpha$: Indeed, by Lemma 16(i), we have $x \,\varepsilon\, y \to \langle x < y\rangle = 1$ and, by Lemma 5, we have: $\langle x < y\rangle = 1 \to \langle \alpha x < \alpha y\rangle = \alpha$. Therefore, if $x, y \,\varepsilon\, X$ and $x \neq y$, we have, for instance $x \,\varepsilon\, y$, therefore $\langle \alpha x < \alpha y\rangle = \alpha$ and therefore $\alpha x \neq \alpha y$ since $\alpha \neq 0$.

Thus, there exists a function with domain $\mathcal{N}_\alpha$ which is surjective onto Δ. The same reasoning, applied to $\neg\alpha$ gives the same result for $\neg\alpha$. But this contradicts Lemma 20. ◀

▶ Remark. Theorem 24 shows that it is impossible to define Von Neumann ordinals in $\mathcal{N}$, with ε instead of $\in$, unless $\daleth 2$ is trivial, i.e. the realizability model is, in fact, a forcing model.

6 The model $\mathcal{M}_\mathcal{D}$

For each formula $F[x_1, \ldots, x_n]$ of ZF, we have defined, in the ground model $\mathcal{M}$, an n-ary function symbol with values in $\{0, 1\}$, denoted by $\langle F[x_1, \ldots, x_n]\rangle$, by setting, for any individuals $a_1, \ldots, a_n$ of $\mathcal{M}$: $\langle F[a_1, \ldots, a_n]\rangle = 1 \Leftrightarrow \mathcal{M} \models F[a_1, \ldots, a_n]$. In $\mathcal{N}$, the function symbol $\langle F[x_1, \ldots, x_n]\rangle$ takes its values in the Boolean algebra $\daleth 2$.

We define, in $\mathcal{N}$, two binary relations $\in_\mathcal{D}$ and $=_\mathcal{D}$, by setting:

$$(x \in_\mathcal{D} y) \equiv \mathcal{D}[\langle x \in y\rangle]; \quad (x =_\mathcal{D} y) \equiv \mathcal{D}[\langle x = y\rangle]\,.$$

The class $\mathcal{N}$, equipped with these relations, will be denoted $\mathcal{M}_\mathcal{D}$.

For each formula $F[\vec{x}, y]$ of ZF, with $n+1$ free variables $x_1, \ldots, x_n, y$, we can define, by means of the *choice principle* in $\mathcal{M}$, an n-ary function symbol f_F, such that:

$$\mathcal{M} \models \forall \vec{x}\,(F[\vec{x}, f_F(\vec{x})] \to \forall y\, F[\vec{x}, y])\ ;$$

f_F is called the *Skolem function* of the formula $F[\vec{x}, y]$.

▶ **Lemma 25.**
(i) $\mathsf{I} \Vdash \forall\vec{x}\forall y\,(\langle\forall y\, F[\vec{x},y]\rangle \leq \langle F[\vec{x},y]\rangle)$
(ii) $\mathsf{I} \Vdash \forall\vec{x}\forall y\,(\langle\forall y\, F[\vec{x},y]\rangle = \langle F[\vec{x},f_F(\vec{x})]\rangle)$.

Proof. Trivial. ◀

For each formula $F[\vec{x}]$ of ZF, we define, by recurrence on F, *a formula of* ZF_ε, which has the same free variables, and that we denote $\mathcal{M}_\mathcal{D} \models F[\vec{x}]$ (read: $\mathcal{M}_\mathcal{D}$ *satisfies* $F[\vec{x}]$).

- F is atomic: $(\mathcal{M}_\mathcal{D} \models x_1 \in x_2)$ is $x_1 \in_\mathcal{D} x_2$; $(\mathcal{M}_\mathcal{D} \models x_1 = x_2)$ is $x_1 =_\mathcal{D} x_2$; $(\mathcal{M}_\mathcal{D} \models \bot)$ is $\bot$.
- $F \equiv F_0 \to F_1$: then $(\mathcal{M}_\mathcal{D} \models F)$ is the formula $(\mathcal{M}_\mathcal{D} \models F_0) \to (\mathcal{M}_\mathcal{D} \models F_1)$.
- $F[\vec{x}] \equiv \forall y\, G[\vec{x},y]$: then $(\mathcal{M}_\mathcal{D} \models F[\vec{x}])$ is the formula $\forall y(\mathcal{M}_\mathcal{D} \models G[\vec{x},y])$.

▶ **Lemma 26.** *For each formula* $F[\vec{x}]$ *of ZF, we have* $\Vdash \forall\vec{x}\Big((\mathcal{M}_\mathcal{D} \models F[\vec{x}]) \leftrightarrow \mathcal{D}\langle F[\vec{x}]\rangle\Big)$.

Proof. By recurrence on the length of F. If F is atomic, we have: $\mathsf{I} \Vdash \forall\vec{x}\Big((\mathcal{M}_\mathcal{D} \models F[\vec{x}]) \to \mathcal{D}\langle F[\vec{x}]\rangle\Big)$ and $\mathsf{I} \Vdash \forall\vec{x}\Big(\mathcal{D}\langle F[\vec{x}]\rangle \to (\mathcal{M}_\mathcal{D} \models F[\vec{x}])\Big)$ because $(\mathcal{M}_\mathcal{D} \models F[\vec{x}])$ is identical with $\mathcal{D}\langle F[\vec{x}]\rangle$.

If $F \equiv F_0 \to F_1$, the formula $(\mathcal{M}_\mathcal{D} \models F) \leftrightarrow \mathcal{D}\langle F\rangle$ is: $((\mathcal{M}_\mathcal{D} \models F_0) \to (\mathcal{M}_\mathcal{D} \models F_1)) \leftrightarrow \mathcal{D}\langle F_0 \to F_1\rangle$. Since $\mathcal{D}$ is an ultrafilter, this formula is equivalent with: $((\mathcal{M}_\mathcal{D} \models F_0) \to (\mathcal{M}_\mathcal{D} \models F_1)) \leftrightarrow (\mathcal{D}\langle F_0\rangle \to \mathcal{D}\langle F_1\rangle)$, which is a logical consequence of: $(\mathcal{M}_\mathcal{D} \models F_0) \leftrightarrow \mathcal{D}\langle F_0\rangle$ and $(\mathcal{M}_\mathcal{D} \models F_1) \leftrightarrow \mathcal{D}\langle F_1\rangle$. Hence the result, by the recurrence hypothesis.

If $F[\vec{x}] \equiv \forall y\, G[\vec{x},y]$, let $f_G(\vec{x})$ be the Skolem function of G. Then, we have $(\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y]) \equiv \forall y(\mathcal{M}_\mathcal{D} \models G[\vec{x},y])$, and therefore: $\mathsf{I} \Vdash (\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y]) \to (\mathcal{M}_\mathcal{D} \models G[\vec{x},f_G(\vec{x})])$. Therefore, by the recurrence hypothesis, we have: $\Vdash (\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y]) \to \mathcal{D}\langle G[\vec{x},f_G(\vec{x})]\rangle$. Applying Lemma 25(ii), we obtain $\Vdash (\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y]) \to \mathcal{D}\langle\forall y\, G[\vec{x},y]\rangle$. Conversely, by Lemma 25(i), we have $\Vdash \forall y\,(\mathcal{D}\langle\forall y\, G[\vec{x},y]\rangle \to \mathcal{D}\langle G[\vec{x},y]\rangle)$. Therefore, applying the recurrence hypothesis, we obtain: $\Vdash \mathcal{D}\langle\forall y\, G[\vec{x},y]\rangle \to \forall y(\mathcal{M}_\mathcal{D} \models G[\vec{x},y])$, and thus, by definition of $(\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y])$: $\Vdash \mathcal{D}\langle\forall y\, G[\vec{x},y]\rangle \to (\mathcal{M}_\mathcal{D} \models \forall y\, G[\vec{x},y])$. ◀

▶ **Theorem 27.** $\mathcal{M}_\mathcal{D}$ *is an elementary extension of the ground model* $\mathcal{M}$.

▶ Remark. Theorem 27 is, in fact, true for *any* ultrafilter on $\daleth 2$, with the same proof.

Proof. Let $F[\vec{a}]$ be a closed formula of ZF, with parameters $a_1,\dots,a_n$ in $\mathcal{M}$. If $\mathcal{M} \models F[\vec{a}]$, we have $\langle F[\vec{a}]\rangle = 1$ (by definition), and therefore, of course, $\Vdash \mathcal{D}\langle F[\vec{a}]\rangle$. Therefore, by Lemma 26, we have $\Vdash (\mathcal{M}_\mathcal{D} \models F[\vec{a}])$. If $\mathcal{M} \not\models F[\vec{a}]$, then $\mathcal{M} \models \neg F[\vec{a}]$; therefore, we have $\Vdash (\mathcal{M}_\mathcal{D} \models \neg F[\vec{a}])$. ◀

▶ **Theorem 28.** *Let* $\sqsubset$ *be a well founded binary relation, defined in the ground model* $\mathcal{M}$. *Then the relation* $\mathcal{D}\langle x \sqsubset y\rangle$ *is well founded in the realizability model* $\mathcal{N}$.

▶ Remark. Theorem 28 is an improvement on Theorem 8.

Notations. We shall write $x \sqsubset_\mathcal{D} y$ for $\langle x \sqsubset y\rangle\, \varepsilon\, \mathcal{D}$. Recall that $x < y$ means $x \in \mathrm{Cl}(y)$; and that $x <_\alpha y$ means $\langle x < y\rangle \geq \alpha$, for $\alpha\, \varepsilon\, \daleth 2$.

We define, in the model $\mathcal{M}$, a binary relation $\sqsubset\!\!\sqsubset$ on the class $\{0,1\}\times\mathcal{M}$ by setting, for any $\alpha,\alpha' \in \{0,1\}$ and a,a' in $\mathcal{M}$:

$$(\alpha',a') \sqsubset\!\!\sqsubset (\alpha,a) \Leftrightarrow (\alpha' < \alpha) \vee (\alpha = \alpha' = 0 \wedge a' < a) \vee (\alpha = \alpha' = 1 \wedge a' \sqsubset a)\,.$$

The relation $\sqsubset$ is the *ordered direct sum* of the relations $\sqsubset, <$. It is easily shown that it is *well founded in* $\mathcal{M}$.

The binary function symbol associated with this relation, of domain $\{0,1\}\times\mathcal{M}$ and values in $\{0,1\}$, is given by:

$$\langle(\alpha',a')\sqsubset(\alpha,a)\rangle = (\neg\alpha'\wedge\alpha)\vee(\neg\alpha'\wedge\neg\alpha\wedge\langle a'<a\rangle)\vee(\alpha'\wedge\alpha\wedge\langle a'\sqsubset a\rangle)\,.$$

This definition gives, in $\mathcal{N}$, a binary function symbol with arguments in $\rrbracket 2\times\mathcal{N}$, and values in $\rrbracket 2$. By Theorem 8, *the binary relation* $\langle(\alpha',a')\sqsubset(\alpha,a)\rangle = 1$ *is well founded in* $\mathcal{N}$.

Proof. By contradiction: we assume that the binary relation $\sqsubset_{\mathcal{D}}$ is not well founded. Thus, there exists a_0, A_0 such that $a_0\,\varepsilon\,A_0$ and A_0 has no minimal ε-element for $\sqsubset_{\mathcal{D}}$. We define, in $\mathcal{N}$, the class $\mathcal{X}$ of ordered pairs (α,x), such that: *There exists* X *such that* $x\,\varepsilon\,X$ *and* X *has no minimal* ε*-element, neither for* $\sqsubset_{\mathcal{D}}$ *nor for* $<_{\neg\alpha}$. Therefore, the formula $\mathcal{X}(\alpha,x)$ is:

$$\alpha\,\varepsilon\,\rrbracket 2\wedge\exists X\left\{x\,\varepsilon\,X,(\forall u\,\varepsilon\,X)\{(\exists v\,\varepsilon\,X)(v\sqsubset_{\mathcal{D}}u),(\exists w\,\varepsilon\,X)(w<_{\neg\alpha}u)\}\right\}.$$

If (α,x) is in $\mathcal{X}$, then we have $\mathcal{D}(\alpha)$: indeed, the set X is non void and has no minimal ε-element for $<_{\neg\alpha}$. Therefore, we have $\neg\mathcal{D}(\neg\alpha)$, and thus $\mathcal{D}(\alpha)$, since $\mathcal{D}$ is an ultrafilter.

We obtain the desired contradiction by showing that the class $\mathcal{X}$ is non void and has no minimal element for the binary relation $\langle(\alpha',x')\sqsubset(\alpha,x)\rangle = 1$.

The ordered pair $(1,a_0)$ is in $\mathcal{X}$: indeed, we have $x<_0 x$ for every x, and therefore A_0 has no minimal ε-element for $<_0$.

Now let (α,a) be in $\mathcal{X}$; we search for (α',a') in $\mathcal{X}$ such that $\langle(\alpha',a')\sqsubset(\alpha,a)\rangle = 1$.

By hypothesis on (α,a), there exists A such that $a\,\varepsilon\,A$ and A has no minimal ε-element, neither for $\sqsubset_{\mathcal{D}}$ nor for $<_{\neg\alpha}$. Thus, there exists $a^0, a^1\,\varepsilon\,A$ such that we have $\mathcal{D}\langle a^0\sqsubset a\rangle$ and $a^1<_{\neg\alpha}a$. We set $\alpha' = (\alpha\wedge\langle a^0\sqsubset a\rangle)$ and therefore, we have $\mathcal{D}(\alpha')$. We set $\beta = \neg\alpha'\wedge\alpha$; therefore $\alpha',\neg\alpha,\beta$ form a partition of 1 in the Boolean algebra $\rrbracket 2$. We have $\neg\,\mathcal{D}(\beta)$; therefore, by definition of $\mathcal{D}$, the relation $<_\beta$ is not well founded. Thus, there exists b, B such that $b\,\varepsilon\,B$ and B has no minimal ε-element for $<_\beta$. Then, we set: $a' = \alpha' a^0\sqcup(\neg\alpha)a^1\sqcup\beta b$ and $A' = \{\alpha' x\sqcup(\neg\alpha)y\sqcup\beta z\ ; x,y\,\varepsilon\,A, z\,\varepsilon\,B\}$.

Therefore, we have $a'\,\varepsilon\,A'$, as needed; moreover: $\neg\alpha'\wedge\neg\alpha\wedge\langle a'<a\rangle = \neg\alpha$, since $\neg\alpha'\geq\neg\alpha$ and $\langle a'<a\rangle\geq\neg\alpha\wedge\langle a^1<a\rangle = \neg\alpha$; $\alpha'\wedge\alpha\wedge\langle a'\sqsubset a\rangle = \alpha'\wedge\langle a'\sqsubset a\rangle = \alpha'\wedge\langle a^0\sqsubset a\rangle = \alpha'$. By definition of $\langle(\alpha',a')\sqsubset(\alpha,a)\rangle$, it follows that $\langle(\alpha',a')\sqsubset(\alpha,a)\rangle = \beta\vee\neg\alpha\vee\alpha' = 1$.

It remains to show that A' has no minimal ε-element for $\sqsubset_{\mathcal{D}}$ and for $<_{\neg\alpha'}$. Therefore, let $u\,\varepsilon\,A'$, thus $u = \alpha' x\sqcup(\neg\alpha)y\sqcup\beta z$ with $x,y\,\varepsilon\,A$ and $z\,\varepsilon\,B$. By hypothesis on A, B, there exists $x',y'\,\varepsilon\,A, x'\sqsubset_{\mathcal{D}}x, y'<_{\neg\alpha}y$ and $z'\,\varepsilon\,B, z'<_\beta z$. Then, if we set $u' = \alpha' x'\sqcup(\neg\alpha)y'\sqcup\beta z'$, we have $u'\,\varepsilon\,A'$. Moreover, we have $\langle u'\sqsubset u\rangle\geq\alpha'\wedge\langle x'\sqsubset x\rangle$, and therefore $\mathcal{D}\langle u'\sqsubset u\rangle$, that is $u'\sqsubset_{\mathcal{D}}u$. Finally, $\langle u'<u\rangle\geq(\neg\alpha\wedge\langle y'<y\rangle)\vee(\beta\wedge\langle z'<z\rangle) = \neg\alpha\vee\beta = \neg\alpha'$; therefore, we have $u'<_{\neg\alpha'}u$. ◀

▶ **Theorem 29.** $\mathcal{M}_{\mathcal{D}}$ *is well founded, and therefore has the same ordinals as* $\mathcal{N}'_{\in}$.

Proof. We apply Theorem 28 to the binary relation $\in$ which is well founded in $\mathcal{M}$. We deduce that the relation $\mathcal{D}\langle x\in y\rangle$, that is $x\in_{\mathcal{D}}y$, is well founded in $\mathcal{N}$. ◀

The relation $\in_{\mathcal{D}}$ is well founded *and extensional*, which means that we have, in $\mathcal{N}$:

$$\forall x\forall y\,(\forall z(z\in_{\mathcal{D}}x\leftrightarrow z\in_{\mathcal{D}}y)\rightarrow\forall z(x\in_{\mathcal{D}}z\rightarrow y\in_{\mathcal{D}}z))\ .$$

It follows that we can define a *collapsing*, by means of a function symbol Φ, which is an *isomorphism* of $(\mathcal{M}_{\mathcal{D}}, \in_{\mathcal{D}})$ on a transitive class in the model $\mathcal{N}_{\in}$ of ZF, which contains the ordinals. This means that we have:

$$\forall x \forall y (y \in_{\mathcal{D}} x \to \Phi(y) \in \Phi(x)) \quad ; \quad \forall x (\forall z \in \Phi(x))(\exists y \in_{\mathcal{D}} x)\, z \simeq \Phi(y)\,.$$

The definition of Φ is analogous with that of the *rank function* already defined for a *transitive* well founded relation. The details will be given in a later version of this paper. Il follows that:

▶ **Theorem 30.** *The realizability model $\mathcal{N}_{\in}$ contains a transitive class, which contains the ordinals and is an elementary extension of the ground model $\mathcal{M}$.*

▶ **Corollary 31.** *The class $L^{\mathcal{M}}$ of constructible sets in $\mathcal{M}$ is an elementary submodel of $L^{\mathcal{N}}$.*

Acknowledgements. I want to thank the referees, especially the first one for numerous pertinent and helpful remarks and suggestions.

References

1 S. Berardi, M. Bezem, and T. Coquand. On the computational content of the axiom of choice. *J. Symb. Logic*, 63(2):600–622, 1998.
2 U. Berger and P. Oliva. Modified bar recursion and classical dependent choice. In Springer, editor, *Proc. Logic Colloquium 2001*, pages 89–107, 2005.
3 H.B. Curry and R. Feys. *Combinatory Logic*. North-Holland, 1958.
4 T. Griffin. A formulæ-as-type notion of control. In *Conf. record 17th A.C.M. Symp. on Principles of Progr. Languages*, 1990.
5 W. Howard. *The formulas–as–types notion of construction*, pages 479–490. Acad. Press, 1980.
6 J.-L. Krivine. Realizability algebras : a program to well order $\mathbb{R}$. *Logical Methods in Computer Science*, 7(3:02):1–47, 2011.
7 J.-L. Krivine. Realizability algebras II : new models of ZF + DC. *Logical Methods in Computer Science*, 8(1:10):1–28, 2012.
8 J.-L. Krivine. Bar recursion in classical realisability : dependent choice and well ordering of $\mathbb{R}$. http://arxiv.org/abs/1502.00112, 2012 (to appear).
9 J.-L. Krivine. Realizability algebras III : some examples. `http://arxiv.org/abs/1210.5065`, 2012 (to appear).
10 T. Streicher. A classical realizability model arising from a stable model of untyped λ-calculus, 2013 (to appear).

An Extensional Kleene Realizability Semantics for the Minimalist Foundation

Maria Emilia Maietti and Samuele Maschio

Dipartimento di Matematica, University of Padova
Via Trieste, 63 – I-35121 Padova, Italy
{maietti,maschio}@math.unipd.it

Abstract

We build a Kleene realizability semantics for the two-level Minimalist Foundation **MF**, ideated by Maietti and Sambin in 2005 and completed by Maietti in 2009. Thanks to this semantics we prove that both levels of **MF** are consistent with the formal Church Thesis **CT**.

Since **MF** consists of two levels, an intensional one, called **mTT**, and an extensional one, called **emTT**, linked by an interpretation, it is enough to build a realizability semantics for the intensional level **mTT** to get one for the extensional one **emTT**, too. Moreover, both levels consists of type theories based on versions of Martin-Löf's type theory.

Our realizability semantics for **mTT** is a modification of the realizability semantics by Beeson in 1985 for extensional first order Martin-Löf's type theory with one universe. So it is formalized in Feferman's classical arithmetic theory of inductive definitions, called $\widehat{ID_1}$. It is called *extensional* Kleene realizability semantics since it validates extensional equality of type-theoretic functions **extFun**, as in Beeson's one.

The main modification we perform on Beeson's semantics is to interpret propositions, which are defined primitively in **MF**, in a proof-irrelevant way. As a consequence, we gain the validity of **CT**. Recalling that **extFun**+ **CT**+ **AC** are inconsistent over arithmetics with finite types, we conclude that our semantics does not validate the Axiom of Choice **AC** on generic types. On the contrary, Beeson's semantics does validate **AC**, being this a theorem of Martin-Löf's theory, but it does not validate **CT**. The semantics we present here seems to be the best approximation of Kleene realizability for the extensional level **emTT**. Indeed Beeson's semantics is not an option for **emTT** since **AC** on generic sets added to it entails the excluded middle.

1998 ACM Subject Classification F.4.1 Computability theory, Lambda calculus and related systems

Keywords and phrases Realizability, Type Theory, formal Church Thesis

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.162

1 Introduction

A foundation for mathematics should be called constructive only if the mathematics arising from it could be considered genuinely computable. One way to show this is to produce a realizability model of the foundation where arbitrary sets are interpreted as data types and functions between them are interpreted as programs. A key example is Kleene's realizability model for first-order Intuitionist Arithmetics validating the formal Church Thesis.

Here we will show how to build a realizability model for the *Minimalist Foundation*, for short **MF**, ideated by Maietti and Sambin in [13] and then completed by Maietti in [9], where it is explicit how to extract programs from its proofs. In particular we show that **MF** is consistent with the Church Thesis, for short **CT**. This result is part of a project to know to what extent **MF** enjoys the same properties as Heyting arithmetics.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 162–186

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

The Minimalist Foundation is intended to constitute a common core among the most relevant constructive and classical foundations. One of its novelties is that it consists of two levels: an intensional level, called **mTT**, which should make evident the constructive contents of mathematical proofs in terms of programs, and an extensional level, called **emTT**, formulated in a language close as much as possible to that of ordinary mathematics. Both intensional and extensional levels of **MF** consist of type systems based on versions of Martin-Löf's type theory with the addition of a primitive notion of propositions: the intensional one is based on [17] and the extensional one on [16]. Actually **mTT** can be considered a *predicative* version of Coquand's Calculus of Constructions [4].

To build a realizability model for the two-level Minimalist Foundation, it is enough to build it for its intensional level **mTT**. Indeed an interpretation for the extensional level **emTT** can be then obtained from an interpretation of **mTT** by composing this with the interpretation of **emTT** in a suitable setoid model of **mTT** as in [9] and analyzed in [11]. Moreover, since the interpretation of **CT** from the extensional level to the intensional one is equivalent to **CT** itself according to [9], a model showing consistency of **mTT** with **CT** can be turned into a model showing consistency of **emTT** with **CT**.

Here, we build a realizability model for **mTT+ CT** by suitably modifying Beeson's realizability semantics [2] for the extensional version of first order Martin-Löf's type theory with one universe [16]. So, as Beeson's semantics our model is based on Kleene realizability semantics of intuitionistic arithmetics and it is formalized in Feferman's classical arithmetic theory of inductive definitions, called $\widehat{ID_1}$ ([5]). The theory $\widehat{ID_1}$ is formulated in the language of second-order arithmetics and it consists of PA (Peano Arithmetic) plus the existence of some (not necessary the least) fix point for positive parameter-free arithmetical operators.

We call our Kleene realizability semantics *extensional* since it validates extensional equality of type-theoretic functions **extFun**, as Beeson's one.

The main modification we perform to Beeson's semantics is to interpret propositions, which are defined primitively in **MF**, in a proof-irrelevant way. More in detail we interpret **mTT**-sets as Beeson interpreted Martin-Löf's sets, propositions are interpreted as trivial quotients of Kleene realizability interpretation of intuitionistic connectives, and the universe of **mTT**-small propositions is interpreted as a suitable quotient of some fix point including all the codes of small propositions by using the technique Beeson adopted to interpret Martin-Löf's universe.

As a consequence in our model we gain the validity of **CT** but we loose the validity of the full Axiom of Choice **AC**. Instead in Beeson's semantics, **AC** is valid, being this a theorem of Martin-Löf's theory, but **CT** is not. All these results follow from the well known fact that **extFun+ CT+ AC** over arithmetics with finite types are inconsistent. Therefore in the presence of **extFun** as in our **emTT**, either one validates **CT** as we do here, or **AC** as in Beeson's semantics. Recalling that the addition of **AC** on generic sets in **emTT** entails the excluded middle, Beeson's semantics is not an option for **emTT**. Therefore the semantics we present here appears to be the best approximation of Kleene realizability for the extensional level **emTT**.

Actually a consistency proof for **emTT** with **CT** could also be obtained by interpreting this theory in the internal theory of Hyland's effective topos [7]. But here we have obtained a proof in a *predicative theory*, whilst classical, as $\widehat{ID_1}$. As a future work we intend to generalize the notion of tripos-to-topos construction in [8] in order to extract the categorical structure behind our realizability interpretation to the final goal of building a predicative effective topos.

2 The Minimalist Foundation

In [9] a two-level formal system, called **Minimalist Foundation**, for short **MF**, is completed following the design advocated in [13]. The two levels of **MF** are both given by a type theory à la Martin-Löf: the intensional level, called **mTT**, is an intensional type theory including aspects of Martin-Löf's one in [17] (and extending the set-theoretic version in [13] with collections), and its extensional level, called **emTT**, is an extensional type theory including aspects of extensional Martin-Löf's one in [16]. Then a quotient model of setoids à la Bishop [3, 6, 1, 19] over the intensional level is used in [9] to interpret the extensional level in the intensional one. A categorical study of this quotient model has been carried on in [11, 10, 12] and related to the construction of Hyland's effective topos [7, 8].

MF was ideated in [13] to be constructive and minimalist, that is compatible with (or interpretable in) most relevant constructive and classical foundations for mathematics in the literature. According to these desiderata, **MF** has the following peculiar features (for a more extensive description see also [14]):

- **MF has two types of entities: sets and collections.** This is a consequence of the fact that a minimalist foundation compatible with most of constructive theories in the literature, among which, for example, Martin-Löf's one in [17], should be certainly predicative and based on intuitionistic predicate logic, including at least the axioms of Heyting arithmetic. For instance it could be a many-sorted logic, such as Heyting arithmetic of finite types [20], where sorts, that we call *types*, include the basic sets we need to represent our mathematical entities. But in order to represent topology in an intuitionistic and predicative way, then **MF** needs to be equipped with two kinds of entities: sets and collections. Indeed, the *power of a non-empty set*, namely the discrete topology over a non-empty set, fails to be a set in a predicative foundation, and it is only a *collection*.
- **MF has two types of propositions.** This is a consequence of the previous characteristic. Indeed the presence of sets and collections, where the latter include the representation of power-collections of subsets, yields to distinguish two types of propositions to remain predicative: those closed under quantifications on sets, called *small propositions* in [9], from those closed under any kind of quantification, called *propositions* in [9]. This distinction is crucial in the definition of "subset of a set" we adopt in **MF**: a subset of a set A is indeed an equivalence class of small propositional functions from A.
- **MF has two types of functions.** As in Coquand's Calculus of Constructions [4], or Feferman's predicative theories [5], in **MF** we distinguish the notion of functional relation from that of type-theoretic function. In particular *in **MF** only type-theoretic functions between two sets form a set, while functional relations between two sets form generally a collection.*

  This restriction is crucial to make **MF** compatible with classical predicative theories as Feferman's predicative theories [5]. Indeed it is well-known that the *addition of the principle of excluded middle* can turn a predicative theory where functional relations between sets form a set, as Aczel's CZF or Martin-Löf's type theory, into an impredicative one where *power-collections become sets*.

2.1 The intensional level of the Minimalist Foundation

Here we describe the intensional level of the Minimalist Foundation in [9], which is represented by a dependent type theory called **mTT**. This type theory is written in the style of Martin-

Löf's type theory [17] by means of the following four kinds of judgements:

$$A\ type\ [\Gamma] \qquad A = B\ type\ [\Gamma] \qquad a \in A\ [\Gamma] \qquad a = b \in A\ [\Gamma]$$

that is the type judgement (expressing that something is a specific type), the type equality judgement (expressing when two types are equal), the term judgement (expressing that something is a term of a certain type) and the term equality judgement (expressing the *definitional equality* between terms of the same type), respectively, all under a context Γ.

The word *type* is used as a meta-variable to indicate four kinds of entities: collections, sets, propositions and small propositions, namely

$$type \in \{col, set, prop, prop_s\}\,.$$

Therefore, in **mTT** types are actually formed by using the following judgements:

$$A\ set\ [\Gamma] \qquad D\ col\ [\Gamma] \qquad \phi\ prop\ [\Gamma] \qquad \psi\ prop_s\ [\Gamma]$$

saying that A is a set, that D is a collection, that ϕ is a proposition and that ψ is a small proposition.

Here, contrary to [9] where capital latin letters are used as meta-variables for all types, we use greek letters ψ, ϕ as meta-variables for propositions, we mostly use capital latin letters A, B as meta-variables for sets and capital latin letters C, D as meta-variables for collections.

As in the intensional version of Martin-Löf's type theory, in **mTT** there are two kinds of equality concerning terms: one is the definitional equality of terms of the same type given by the judgement

$$a = b \in A\ [\Gamma]$$

which is decidable, and the other is the propositional equality written

$$\mathsf{Id}(A, a, b)\ prop\ [\Gamma]$$

which is not necessarily decidable.

We now proceed by briefly describing the various kinds of types in **mTT**, starting from small propositions and propositions and then passing to sets and finally collections.

Small propositions in **mTT** include all the logical constructors of intuitionistic predicate logic with equality and quantifications restricted to sets:

$$\phi\ prop_s \ \equiv \bot \mid \ \phi \wedge \psi \ \mid \phi \vee \psi \ \mid \ \phi \to \psi \ \mid (\forall x \in A)\ \phi(x) \ \mid (\exists x \in A)\ \phi(x) \ \mid \ \mathsf{Id}(A, a, b)$$

provided that A is a set.

Then, *propositions* in **mTT** include all the logical constructors of intuitionistic predicate logic with equality and quantifications on all kinds of types, i.e. sets and collections. Of course, small propositions are also propositions.

$$\phi\ prop \ \equiv \phi\ prop_s \ \mid \phi \wedge \psi \mid \phi \vee \psi \mid \ \phi \to \psi \mid (\forall x \in D)\ \phi(x) \mid (\exists x \in D)\ \phi(x) \mid \ \mathsf{Id}(D, d, b)\,.$$

In order to close sets under comprehension, for example to include the set of positive natural numbers $\{x \in \mathrm{N} \mid x \geq 1\}$, and to define operations on such sets, we need to think of propositions as types of their proofs: small propositions are seen as sets of their proofs while generic propositions are seen as collections of their proofs. That is, we add to **mTT** the following rules

$$\textbf{prop}_s\textbf{-into-set)}\ \frac{\phi\ prop_s}{\phi\ set} \qquad\qquad \textbf{prop-into-col)}\ \frac{\phi\ prop}{\phi\ col}$$

Before explaining the difference between the notion of set and collection we describe their constructors in **mTT**.

Sets in **mTT** are characterized as inductively generated types and they include the following:

$$A\ set\ \equiv \phi\ prop_s \mid N_0 \mid N_1 \mid N \mid List(A) \mid (\Sigma x \in A)\, B(x) \mid A + B \mid (\Pi x \in A)\, B(x)$$

where the notation N_0 stands for the empty set, N_1 for the singleton set, N for the set of natural numbers, $List(A)$ for the set of Lists on the set A, $(\Sigma x \in A)B(x)$ for the indexed sum of the family of sets $B(x)$ *set* $[x \in A]$ indexed on the set A, $A + B$ for the disjoint sum of the set A with the set B, $(\Pi x \in A)B(x)$ for the product type of the family of sets $B(x)$ *set* $[x \in A]$ indexed on the set A.

It is worth noting that the set N of the natural numbers is not present in a primitive way in **mTT** in [9] since its rules can be derived by putting $N \equiv List(N_1)$. Here we add it to the syntax of **mTT** because it plays a prominent role in realizability and we want to interpret it directly in $\widehat{ID_1}$ to avoid complications due to list encodings.

Finally, *collections* in **mTT** include the following types:

$$D\ col\ \equiv A\ set\ \mid\ \phi\ prop\ \mid\ \mathsf{prop_s}\ \mid\ A \to \mathsf{prop_s}\ \mid\ (\Sigma x \in D)\, E(x)$$

and all sets are collections thanks to the following rule:

$$\textbf{set-into-col)}\quad \frac{A\ set}{A\ col}$$

where $\mathsf{prop_s}$ stands for the collection of (codes for) small propositions and $A \to \mathsf{prop_s}$ for the collection of propositional functions of the set A, while $(\Sigma x \in D)\, E(x)$ stands for the indexed sum of the family of collections $E(x)$ *col* $[x \in D]$ indexed on the collection D.

Note that the collection of small propositions $\mathsf{prop_s}$ is defined here with codes à la Tarski as in [17], contrary to the version in [9], to make the interpretation easier to understand. Its rules are the following.
Elements of the collection of small propositions are generated as follows:

$$\mathrm{Pr}_1)\ \ \widehat{\bot} \in \mathsf{prop_s} \qquad \mathrm{Pr}_5)\ \frac{A\ set \quad a \in A \quad b \in A}{\widehat{\mathsf{Id}}(A, a, b) \in \mathsf{prop_s}}$$

$$\mathrm{Pr}_2)\ \frac{p \in \mathsf{prop_s} \quad q \in \mathsf{prop_s}}{p \widehat{\vee} q \in \mathsf{prop_s}} \qquad \mathrm{Pr}_6)\ \frac{p(x)\ \mathsf{prop_s} \quad [x \in B] \qquad B\ set}{\widehat{(\exists x \in B)p(x)} \in \mathsf{prop_s}}$$

$$\mathrm{Pr}_3)\ \frac{p \in \mathsf{prop_s} \qquad q \in \mathsf{prop_s}}{p \widehat{\rightarrow} q \in \mathsf{prop_s}} \qquad \mathrm{Pr}_7)\ \frac{p(x) \in \mathsf{prop_s}\ [x \in B] \qquad B\ set}{\widehat{(\forall x \in B)p(x)} \in \mathsf{prop_s}}$$

$$\mathrm{Pr}_4)\ \frac{p \in \mathsf{prop_s} \qquad q \in \mathsf{prop_s}}{p \widehat{\wedge} q \in \mathsf{prop_s}}$$

Elements of the collection of small propositions can be decoded as small propositions via an operator as follows

$$\tau\text{-Pr})\ \frac{p \in \mathsf{prop_s}}{\tau(p)\ prop_s}$$

and this operator satisfies the following definitional equalities:

$$\text{eq-Pr}_1)\ \ \tau(\widehat{\bot}) = \bot\, prop_s \qquad \text{eq-Pr}_5)\ \ \frac{A\ set \quad a \in A \quad b \in A}{\tau(\widehat{\mathsf{Id}}(A,a,b)) = \mathsf{Id}(A,a,b)\, prop_s}$$

$$\text{eq-Pr}_2)\ \ \frac{p \in \mathsf{prop_s} \quad q \in \mathsf{prop_s}}{\tau(p\widehat{\vee}q) = \tau(p) \vee \tau(q)\, prop_s} \qquad \text{eq-Pr}_6)\ \ \frac{p(x)\ \mathsf{prop_s} \quad [x \in B] \qquad B\ set}{\tau(\widehat{(\exists x \in B)p(x)}) = (\exists x \in B)\tau(p(x))\, prop_s}$$

$$\text{eq-Pr}_3)\ \ \frac{p \in \mathsf{prop_s} \qquad q \in \mathsf{prop_s}}{\tau(p\widehat{\to}q) = \tau(p) \to \tau(q)\, prop_s} \qquad \text{eq-Pr}_7)\ \ \frac{p(x) \in \mathsf{prop_s}\ [x \in B] \qquad B\ set}{\tau(\widehat{(\forall x \in B)p(x)}) = (\forall x \in B)\tau(p(x))\, prop_s}$$

$$\text{eq-Pr}_4)\ \ \frac{p \in \mathsf{prop_s} \qquad q \in \mathsf{prop_s}}{\tau(p\widehat{\wedge}q) = \tau(p) \wedge \tau(q)\, prop_s}$$

In the realizability interpretation of **mTT** we need to define a subset of natural numbers including codes of **mTT**-sets in order to define the subset of codes of small propositions closed under quantification on sets. The existence of such a subset of set codes says that the realizability interpretation is actually interpreting an extension of **mTT** with a collection of sets. In order to simplify the definition of the realizability interpretation, we interpret an extension of **mTT**, which we call **mTT**^{s}, with the addition of the collection Set of set codes whose related rules are the following. We don't give any elimination and conversion rule as those of universes à la Tarski in [17] since it would not be validated in the model (because we do not have least fix-points in $\widehat{ID_1}$).

Collection of sets

F-Se) $\mathsf{Set}\ col$

Elements of the collection of sets are generated as follows:

$$\text{Se}_e)\ \ \widehat{N_0} \in \mathsf{Set} \qquad \text{Se}_s)\ \ \widehat{N_1} \in \mathsf{Set} \qquad \text{Se}_n)\ \ \widehat{N} \in \mathsf{Set}$$

$$\text{Se}_l)\ \ \frac{a \in \mathsf{Set}}{\widehat{List}(a) \in \mathsf{Set}} \qquad \text{Se}_u)\ \ \frac{a \in \mathsf{Set} \quad b \in \mathsf{Set}}{a\widehat{+}b \in \mathsf{Set}}$$

$$\text{Se}_\Sigma)\ \ \frac{a(x)\ \mathsf{Set} \quad [x \in B] \qquad B\ set}{\widehat{(\Sigma x \in B)a(x)} \in \mathsf{Set}} \qquad \text{Se}_\Pi)\ \ \frac{a(x)\ \mathsf{Set} \quad [x \in B] \qquad B\ set}{\widehat{(\Pi x \in B)a(x)} \in \mathsf{Set}}$$

$$\text{sp-i-s})\ \ \frac{p \in \mathsf{prop_s}}{p \in \mathsf{Set}}$$

mTT can be viewed as a *predicative version* of the Calculus of Constructions [4], for short CoC. The main difference with respect to CoC is that **mTT** distinguishes between sets and collections in a way similar to the distinction between sets and classes in axiomatic set theory. However, all types of **mTT**, i.e. small propositions, propositions, sets and collections, are predicative entities in the sense that their elements can be generated in an inductive way by a finite number of rules. According to the notion of set in Bishop [3] and Martin-Löf [15], all **mTT**-types are actually sets, and in fact **mTT**-types can be interpreted as sets in the intensional version of Martin-Löf's type theory in [17]. The **mTT**-distinction between sets and collections, and the corresponding distinction between small propositions and propositions, is motivated by the need of distinguishing between predicative entities whose notion of element is a closed concept, and these are called sets, and those entities whose notion of element is an open concept, and these are called collections. The motivating idea is that a set is inductively generated by a finite number of rules whose associated inductive principle does not vary when the theory **mTT** is extended with new entities (sets, collections

or propositions). On the contrary a collection is inductively generated by a finite number of rules which may vary when the theory is extended with new entities. Typical examples of collections are universes (of sets or propositions): if we extend the theory **mTT** with a new small proposition, then we need to add a new rule inserting this new small proposition in the collection of small propositions.

We recall from [13] that the distinction between propositions and sets is crucial to avoid the validity of choice principles.

Finally, it is worth noting that in **mTT** we restrict substitution term equality rules to explicit substitution term equality rules of the form

$$\text{sub)}\ \frac{\begin{array}{c} c(x_1,\ldots,x_n) \in C(x_1,\ldots,x_n)\ \ [\,x_1 \in A_1,\ \ldots,\ x_n \in A_n(x_1,\ldots,x_{n-1})\,] \\ a_1 = b_1 \in A_1\ \ \ldots\ \ a_n = b_n \in A_n(a_1,\ldots,a_{n-1}) \end{array}}{c(a_1,\ldots,a_n) = c(b_1,\ldots,b_n) \in C(a_1,\ldots,a_n)}$$

in place of usual term equality rules preserving term constructions typical of Martin-Löf's type theory in [17]. This restriction, and in particular the absence of the so called ξ-rule of lambda-terms

$$\xi\ \frac{c = c' \in C\ [x \in B]}{\lambda x^B.c = \lambda x^B.c' \in (\Pi x \in B)C}$$

seems to be crucial to prove consistency of **mTT** with **AC**+**CT**, as advocated in [13], by means of a realizability semantics à la Kleene, but this is still an open problem (the realizability semantics given here does not help to solve this since it can not validate **AC** on all types). It is worth to recall from [9] that our restriction of term equality does not affect the possibility of adopting **mTT** as the intensional level of a two-level constructive foundation as intended in [13]. Indeed the term equality rules of **mTT** suffice to interpret an extensional level including extensional equality of functions, as that represented by **emTT**, by means of the quotient model described in [9] and studied abstractly in [11, 10, 12].

2.2 The extensional level of the Minimalist Foundation

Here we briefly describe the extensional level **emTT** of the Minimalist Foundation. This is an extensional dependent type theory extending extensional Martin-Löf's type theory in [16] with primitive (proof-irrelevant) propositions, power-collections and quotients.

The rules of **emTT** are formulated by using the same kinds of judgements used for **mTT**. The main peculiar characteristics of **emTT** in comparison to **mTT** are the following.

1. A primary difference between **emTT** and **mTT** is the usual difference between the so called intensional version of Martin-Löf's type theory [17] and its extensional one in [16] and this is the fact that the definitional equality of terms

   $$a = b \in A\ [\Gamma]$$

   is no longer *decidable* in **emTT** as it is in the intensional **mTT**. This is in turn due to the fact that the propositional equality of **emTT** as that of [16], called $\mathsf{Eq}(A,a,b)$, is extensional in the sense that the provability of $\mathsf{Eq}(A,a,b)$ $[\Gamma]$ in **emTT** is equivalent to the derivation of the judgement $a = b \in A$ $[\Gamma]$. Instead, in **mTT** only the derivation of the definitional equality judgement $a = b \in A$ $[\Gamma]$ implies internally the provability of the intensional propositional equality $\mathsf{Id}(A,a,b)$ $[\Gamma]$ under a generic context.

2. Another peculiar feature of **emTT** employs the distinction between propositions and sets: this is the addition of proof-irrelevance for propositions captured by the following rules

$$\textbf{prop-mono})\ \frac{\phi\ prop\ [\Gamma] \qquad p \in \phi\ [\Gamma] \quad q \in \phi\ [\Gamma]}{p = q \in \phi\ \ [\Gamma]} \qquad \textbf{prop-true})\ \frac{\phi\ prop \qquad p \in \phi}{\mathsf{true} \in \phi}$$

saying that a proof of a proposition is *unique* and equal to a canonical proof term called true. Of course, these rules can not be added to an extensional theory identifying propositions with sets as Martin-Löf's one in [16], because they would trivialize all constructors. Moreover, these rules are not present in the intensional level **mTT** because proof-irrelevance is a typical extensional condition. Indeed, **emTT**-propositions can be thought of as quotients of intensional propositions under the trivial equivalence relation between proofs.

3. Other key differences between the type theories **mTT** and **emTT** are the addition in **emTT** of quotient sets

$$A/\rho\ set\ [\Gamma]$$

provided that ρ is a small equivalence relation $\rho\ prop_s\ [x \in A, y \in A]$ on the set A, and the addition of the power-collection of the singleton and of the power-collection of a generic set A

$$\mathcal{P}(1) \qquad\qquad A \to \mathcal{P}(1)$$

4. A further difference between the type theories **mTT** and **emTT** concerns the equality rules between terms. Indeed in **emTT** equality rules between terms are the usual ones typical of an extensional type theory in [16] preserving all term constructors. In particular, equality of lambda-functions is extensional, namely it is possible to prove

$$(\forall x \in A)\mathsf{Eq}(\, B(x), f(x)\,,\, g(x)) \ \to\ \mathsf{Eq}(\, (\Pi x \in A)B(x)\,,\ \lambda x.f(x)\ ,\ \lambda x.g(x)\,)$$

This proposition is not necessarily provable at the intensional level **mTT** when substituting the extensional propositional equality $\mathsf{Eq}(A, a, b)$ with the intensional one $\mathsf{Id}(A, a, b)$.

We end by recalling from [9] that *a model for **mTT** can be turned into a model for **emTT** by using the interpretation of **emTT** into **mTT** described in* [9]. Therefore in the following we are going to define a realizability interpretation just for **mTT**, to get one also for **emTT**.

2.3 Untyped syntax of mTT^s

Usually in type theory the syntax is introduced *in fieri*; for example terms are introduced typically after deriving some conditions or constraints which are required to define them. However for semantical purposes it looks more convenient to present the syntax *a priori in a partial way* by eliminating parts of usual restrictions.

Therefore, since we want to define a realizability interpretation for **mTT**^{s}, we introduce here the syntax of all **mTT**^{s}-type and term constructors in a partial way and we refer the reader to look at [9] for all the **mTT**-rules. Then we will define a partial interpretation for terms of our *extended* syntax and check that this interpretation is well defined in case the constraints for introducing them are validated by the model.

▶ **Definition 1.** Let $[\underline{x}]$ be a context, i. e. $[\underline{x}] = [x_1, ..., x_n]$ is a possibly empty list of distinct variables. *Terms, small propositions, sets, propositions and collections* in context are defined according to the following conditions. If

1. $t\,[\underline{x}]\,, t'\,[\underline{x}]\,, t''\,[\underline{x}]\,, s\,[\underline{x}, y]\,, s'\,[\underline{x}, y]\,, r\,[\underline{x}, y, z]\,, q\,[\underline{x}, y, z, u]$ are terms in context;
2. $\phi\,[\underline{x}]\,, \phi'\,[\underline{x}]\,, \psi\,[\underline{x}, y]$ are small propositions in context;
3. $A\,[\underline{x}]\,, A'\,[\underline{x}]\,, B\,[\underline{x}, y]$ are sets in context;
4. $\eta\,[\underline{x}]\,, \eta'\,[\underline{x}]\,, \rho\,[\underline{x}, y]$ are propositions in context;
5. $D\,[\underline{x}]\,, E\,[\underline{x}, y]$ are collections in context,

then

1. $x_i\,[\underline{x}]$ is a term in context;
the empty set eliminator $\mathsf{emp}_0(t)\,[\underline{x}]$ is a term in context;
the singleton constant $\star\,[\underline{x}]$ and *the singleton eliminator* $El_{N_1}(t, t')\,[\underline{x}]$ are terms in context;
the zero constant $0\,[\underline{x}]$, *the successor constructor* $\mathsf{succ}(t)\,[\underline{x}]$ and *the eliminator of natural numbers* $El_N(t, t', (y, z)r)\,[\underline{x}]$ are terms in context[1];
the lambda abstraction of dependent product $\lambda y.s\,[\underline{x}]$ and *its application* $\mathsf{Ap}(t, t')\,[\underline{x}]$ are terms in context;
the pairing of strong indexed sum $\langle t, t'\rangle\,[\underline{x}]$ and its eliminator $El_\Sigma(t, (y, z)r)\,[\underline{x}]$ are terms in context;
the first injection of binary disjoint sum $\mathsf{inl}(t)\,[\underline{x}]$, *its second injection* $\mathsf{inr}(t)\,[\underline{x}]$ and *its eliminator* $El_+(t, (y)s, (y)s')\,[\underline{x}]$ are terms in context;
the empty list constant $\epsilon\,[\underline{x}]$, *the list constructor* $\mathsf{cons}(t, t')\,[\underline{x}]$ and *the list eliminator* $El_{List}(t, t', (y, z, u)q)\,[\underline{x}]$ are terms in context;
the false eliminator $\mathsf{r}_0(t)\,[\underline{x}]$ is a term in context;
the pairing of conjunction $\langle t,_\wedge t'\rangle\,[\underline{x}]$, and *its first and second projections* $\pi_1^\wedge(t)\,[\underline{x}]$ and $\pi_2^\wedge(t)\,[\underline{x}]$ are terms in context;
the first injection of disjunction $\mathsf{inl}_\vee(t)\,[\underline{x}]$, *the second injection of disjunction* $\mathsf{inr}_\vee(t)\,[\underline{x}]$ and *its eliminator* $El_\vee(t, (y)s, (y)s')\,[\underline{x}]$ are terms in context;
the lambda abstraction of implication $\lambda_\to y.s\,[\underline{x}]$ and *its application* $\mathsf{Ap}_\to(t, t')\,[\underline{x}]$ are terms in context;
the pairing of existential quantification $\langle t,_\exists t'\rangle\,[\underline{x}]$ and *its eliminator* $El_\exists(t, (y, z)r)\,[\underline{x}]$ are terms in context;
the lambda abstraction of universal quantification $\lambda_\forall y.s\,[\underline{x}]$ and *its application* $\mathsf{Ap}_\forall(t, t')\,[\underline{x}]$ are terms in context;
the Propositional Identity constructor $\mathsf{id}(t)\,[\underline{x}]$ and *its eliminator* $El_{\mathsf{Id}}(t, t', t'', (y)s)\,[\underline{x}]$ [2] are terms in context;
the empty set code $\widehat{N_0}[\underline{x}]$, *the singleton code* $\widehat{N_1}[\underline{x}]$, *the natural numbers set code* $\widehat{N}[\underline{x}]$, *the dependent product code* $\widehat{(\Pi y \in A)s}[\underline{x}]$, *the dependent sum code* $\widehat{(\Sigma y \in A)s}[\underline{x}]$, *the disjoint sum code* $\widehat{t + t'}[\underline{x}]$, *the list code* $\widehat{List}(t)[\underline{x}]$, *the falsum code* $\widehat{\bot}$, *the conjunction code* $\widehat{t \wedge t'}$, *the disjunction code* $\widehat{t \vee t'}$, *the implication code* $\widehat{t \to t'}$, *the existential quantification code* $\widehat{(\exists y \in A)s}\,[\underline{x}]$, *the universal quantification code* $\widehat{(\forall y \in A)s}\,[\underline{x}]$ and *the propositional identity code* $\widehat{\mathsf{Id}}(A, t, t')\,[\underline{x}]$ are terms in context;

2. $\bot\,[\underline{x}]$ and $\tau(t)\,[\underline{x}]$ are small propositions in context;
$\phi \wedge \phi'\,[\underline{x}]$, $\phi \vee \phi'\,[\underline{x}]$ and $\phi \to \phi'\,[\underline{x}]$ are small propositions in context;
$(\exists y \in A)\,\psi\,[\underline{x}]$, $(\forall y \in A)\,\psi\,[\underline{x}]$ and $\mathsf{Id}(A, t, t')\,[\underline{x}]$ are small propositions in context;

---

[1] The rules for these constructors derive from those of $List(N_1)$ in **mTT** by identifying 0 with ϵ, $\mathsf{succ}(t)$ with $\mathsf{cons}(t, \star)$ and $El_N(t, t', (y, z)r)$ with $El_{List(N_1)}(t, t', (y, y', z)r)$.
[2] In the rules for $\mathsf{Id}(A, a, b)$ of **mTT** the eliminator $El_{\mathsf{Id}}(p, (x)c)$ is substituted by an eliminator $El_{\mathsf{Id}}(a, b, p, (x)c)$ with explicit reference to $a \in A$ and $b \in A$. The rules remain the same.

3. $\phi\,[\underline{x}]$ is a set in context;
   $N_0\,[\underline{x}]\,, N_1\,[\underline{x}]$ and $N\,[\underline{x}]$ are sets in context;
   $(\Pi y \in A)\,B\,[\underline{x}]$, $(\Sigma y \in A)\,B\,[\underline{x}]$, $A + A'\,[\underline{x}]$ and $List(A)\,[\underline{x}]$ are sets in context;

4. $\phi\,[\underline{x}]$ is a proposition in context;
   $\eta \wedge \eta'\,[\underline{x}]$, $\eta \vee \eta'\,[\underline{x}]$ and $\eta \to \eta'\,[\underline{x}]$ are propositions in context;
   $(\exists y \in D)\,\rho\,[\underline{x}]$ and $(\forall y \in D)\,\rho\,[\underline{x}]$ and $\mathsf{Id}(D, t, t')\,[\underline{x}]$ are propositions in context;

5. $\eta\,[\underline{x}]$ and $A\,[\underline{x}]$ are collections in context;
   $\mathsf{Set}\,[\underline{x}]$ is a collection in context;
   $\mathsf{prop_s}\,[\underline{x}]$ and $A \to \mathsf{prop_s}\,[\underline{x}]$ are collections in context;
   $(\Sigma y \in D)E\,[\underline{x}]$ is a collection in context.

For sets in context $A\,[\underline{x}]$ we define an abbreviation $\widehat{A}\,[\underline{x}]$ as follows:
1. $\widehat{\bot}$, $\widehat{N_0}$, $\widehat{N_1}$ and $\widehat{N}$ were already defined;
2. $\widehat{((\Pi y \in A)\,B)} = (\widehat{\Pi y \in A})\,\widehat{B}$, $\widehat{((\Sigma y \in A)\,B)} = (\widehat{\Sigma y \in A})\,\widehat{B}$,
3. $\widehat{A + A'} = \widehat{A} \widehat{+} \widehat{A'}$, $\widehat{List(A)} = \widehat{List}(\widehat{A})$,
4. $\widehat{\phi \wedge \phi'} = \widehat{\phi} \,\widehat{\wedge}\, \widehat{\phi'}$, $\widehat{\phi \vee \phi'} = \widehat{\phi} \,\widehat{\vee}\, \widehat{\phi'}$, $\widehat{\phi \to \phi'} = \widehat{\phi} \,\widehat{\to}\, \widehat{\phi'}$,
5. $\widehat{((\exists y \in A)\,\psi)} = (\widehat{\exists y \in A})\,\widehat{\psi}$, $\widehat{((\forall y \in A)\,\psi)} = (\widehat{\forall y \in A})\,\widehat{\psi}$, $\widehat{\mathsf{Id}(A, t, s)} = \widehat{\mathsf{Id}}\,(A, t, s)$,
6. $\widehat{\tau(t)} = t$.

It is clear that the previous definition is overabundant with respect to the common use in type theory. We introduced some terms which we will never find in any standard type theory, as for example the term $0\widehat{\wedge}El_{N_1}(\lambda x.x, \lambda_{\to} y.y)$ which is obtained by gluing together terms which usually have types which are not compatible. For example 0 is usually typed as a natural number, while $\widehat{\wedge}$ connects codes for small propositions.

3 The realizability interpretation for mTT^s

3.1 The system $\widehat{ID_1}$

The preliminary step in the presentation of the Kleene realizability interpretation consists in presenting the theory of *Inductive Definitions* $\widehat{ID_1}$ in which we will interpret **mTT**^{s}. The system $\widehat{ID_1}$ is a predicative fragment of second-order arithmetic, more precisely it is the predicative fragment of second-order arithmetic extending Peano arithmetics with some (not necessarily least) fix points for each positive arithmetical operator. Its number terms are number variables (we assume that these variables are equal to those of **mTT**^{s}), the constant 0 and the terms built by applying the unary successor functional symbol *succ* and the binary sum and product functional symbols $+$ and $*$ to number terms. Set terms are only set variables $X, Y, Z....$ The *arithmetical* formulas are obtained starting from $t = s$ and $t\varepsilon X$ with t, s number terms and X a set variable, by applying the connectives $\wedge, \vee, \neg, \to$ and the number quantifiers $\forall x$, $\exists x$. Moreover let us give the following two definitions.

▶ **Definition 2.** An occurrence of a set variable X in an arithmetical formula φ is *positive* or *negative* according to the following conditions:
1. it is positive if φ is $t\varepsilon X$ for some number term t,
2. it is positive (negative) if φ is $\psi \wedge \psi'$, $\psi' \wedge \psi$, $\psi \vee \psi'$, $\psi' \vee \psi$, $\psi' \to \psi$, $\exists x\,\psi$ or $\forall x\,\psi$ and it is a positive (negative) occurrence of X in ψ or φ is $\psi \to \psi'$ or $\neg\psi$ and the occurrence of X is a negative (positive) occurrence of X in ψ.

▶ **Definition 3.** An arithmetical formula φ with exactly one free number variable x and one free set variable X which occurs only positively is called an *admissible* formula.
In order to define the system $\widehat{ID_1}$ we add to the language of arithmetic a unary predicate symbol P_φ for every admissible formula φ . The atomic formulas of $\widehat{ID_1}$ are
1. $t = s$ with t and s number terms,
2. $t\varepsilon X$ with t a number term and X a set variable,
3. $P_\varphi(t)$ with t a number term and φ an admissible formula.

All formulas of $\widehat{ID_1}$ are obtained by atomic formulas by applying connectives, number quantifiers and set quantifiers.

The axioms of $\widehat{ID_1}$ are the axioms of Peano Arithmetic plus the following three axiom schemata:
1. *Comprehension schema*: for all formulas $\varphi(x)$ of $\widehat{ID_1}$ without set quantifiers

   $$\exists X \forall x (x\varepsilon X \leftrightarrow \varphi(x))$$

2. *Induction schema*: for all formulas $\varphi(x)$ of $\widehat{ID_1}$

   $$(\varphi(0) \wedge \forall x(\varphi(x) \rightarrow \varphi(succ(x)))) \rightarrow \forall x\, \varphi(x)$$

3. *Fix point schema*: for all admissible formulas φ

   $$\varphi[P_\varphi/X] \leftrightarrow P_\varphi(x)$$

   where $\varphi[P_\varphi/X]$ is the result of substituting in φ every atomic formula $t\varepsilon X$ with $P_\varphi(t)$.

The system $\widehat{ID_1}$ allows us to define predicates as fix points, by using axiom schema 3, if they are presented in a appropriate way (i. e. using admissible formulas).

A *definable class* $\mathcal{C}$ of $\widehat{ID_1}$ is a formal writing $\{x|\varphi(x)\}$ where $\varphi(x)$ is a formula of $\widehat{ID_1}$. In this case we write $x\varepsilon\mathcal{C}$ as a shorthand for $\varphi(x)$.

Notation of computable operators in $\widehat{ID_1}$

As it is well known, it is certainly possible to express a Gödelian coding of recursive functions in $\widehat{ID_1}$ using Kleene's predicate since it is already possible to do this in PA. In particular we can consider a definitional extension of $\widehat{ID_1}$ (which we still call $\widehat{ID_1}$) in which there are first-order terms with Kleene's brackets $\{t\}(s)$ and there is a predicate $\{t\}(s)\downarrow$ stating that the term with Kleene's brackets is well defined (s is in the domain of the recursive function coded by t). We will write $\{t\}(s_1, ..., s_\mathsf{n})$ as a shorthand defined by induction: it is $\{t\}(s_1)$ if $n = 1$ while if $n > 1$ and if we have already defined $\{t\}(s_1, ..., s_\mathsf{n})$, then $\{t\}(s_1, ..., s_{\mathsf{n}+1}) = \{\{t\}(s_1, ..., s_\mathsf{n})\}(s_{\mathsf{n}+1})$. We denote by **succ** a numeral for which $\{\mathbf{succ}\}(x) = succ(x)$ in $\widehat{ID_1}$.

As we well know, the s-m-n lemma (see e. g. [18]) gives the structure of a partial combinatorial algebra to natural numbers endowed with Kleene application and this structure can be expressed in $\widehat{ID_1}$. In particular we can find numerals $\mathbf{p}, \mathbf{p}_1, \mathbf{p}_2$ representing a fixed primitive recursive bijective pairing function with primitive recursive first and second projections. We will write $p_1(x)$, $p_2(x)$ and $\langle x, y\rangle$ as abbreviations for $\{\mathbf{p}_1\}(x)$, $\{\mathbf{p}_2\}(x)$ and $\{\mathbf{p}\}(x, y)$ respectively. It is also possible to define a numeral **ite**[3] representing the definition by cases

[3] if then else

($\{\mathbf{ite}\}(n,m,l) \simeq {}^4 m$ if $n = 0$, $\{\mathbf{ite}\}(n,m,l) \simeq l$ if $n \neq 0$). We can also encode recursively finite list of natural numbers with natural numbers in such a way that the empty list is coded by 0 and the concatenation is a recursive function which can be coded by a numeral **cnc**. We have moreover numerals **rec** and **listrec** representing natural numbers recursion and lists recursion. These numbers in particular satisfy the following requirements:

1. $\{\mathbf{rec}\}(n,m,0) \simeq n$;
2. $\{\mathbf{rec}\}(n,m,k+1) \simeq \{m\}(k,\{\mathbf{rec}\}(n,m,k))$;
3. $\{\mathbf{listrec}\}(n,m,0) \simeq n$;
4. $\{\mathbf{listrec}\}(n,m,\mathbf{cnc}(k,l)) \simeq \{m\}(k,l,\{\mathbf{listrec}\}(n,m,k))$.

For this representation of lists, the component functions $(-)_j$, turn out to be recursive.

Moreover we can always define λ-terms $\Lambda x.t$ in $\widehat{ID_1}$ for terms t built with numerals, variables and Kleene application, in such a way that $\{\Lambda x.t\}(n) \simeq t[n/x]$ and it holds that $\{\Lambda x_1 ... \Lambda x_{\mathsf{n}}.t\}(n) \simeq \Lambda x_2 ... \Lambda x_{\mathsf{n}}.t[n/x_1]$; moreover if all variables of t are among $x_1, ..., x_{\mathsf{k}}$, then there is a numeral $\mathbf{n}$ for which $\widehat{ID_1} \vdash \Lambda x_1 ... \Lambda x_{\mathsf{k}}.t = \mathbf{n}$.

3.2 The definition of interpretation

The realizability interpretation for $\mathbf{mTT}^s$ we are going to describe is a modification of Beeson's realizability semantics [2] for the extensional version of first order Martin-Löf's type theory with one universe [16]. So it will be given in $\widehat{ID_1}$ as Beeson's one. Here we describe the key points of such an interpretation on which we follow Beeson's semantics:

- all types of $\mathbf{mTT}^s$ are interpreted as quotients of definable classes of $\widehat{ID_1}$, intended as classes of "their realizers". In particular we use Beeson's technique of interpreting Martin-Löf's universe to interpret the collection of (codes for) small propositions of $\mathbf{mTT}^s$. In order to do this it is crucial to have fix points and hence this is why we work in the theory $\widehat{ID_1}$;
- terms are interpreted as (codes) of recursive functions;
- equality between terms in context is interpreted as extensional equality of recursive functions;
- the interpretation of substitution will be proven to be equivalent to the substitution in interpretation;
- we interpret λ-abstraction by using *s-m-n* lemma of computability, but then, in order to validate the condition of the previous point, we impose equality of type-theoretic functions to be extensional. Therefore the principle of Extensional Equality of Functions will turn out to be valid in our model.

Instead we do not follow Beeson's semantics in the interpretation of propositions:

- in order to validate **formal Church Thesis** we interpret propositions as trivial[5] quotients of original Kleene realizability. As a consequence Martin-Löf's isomorphism of propositions-as-sets together with the validity of the **Axiom of Choice** is not validated in our realizability semantics contrary to Beeson's one.

[4] $a \simeq b$ means that $a\downarrow$ if and only if $b\downarrow$ and in this case $a = b$ in $\widehat{ID_1}$.
[5] A quotient is trivial if it is determined by a trivial relation i. e. a relation for which all pairs of elements are equivalent.

We can summarize the interpretation of terms and types with the following table:

Terms	(codes) of recursive functions
Collections	Quotients of definable classes $(C, \simeq)$
Propositions	quotients of definable classes on trivial $\simeq$

The interpretation of terms

Before giving the interpretation of $\mathbf{mTT}^s$-terms, we need to present explicitly a convention about how to encode $\mathbf{mTT}^s$-sets with numerals. We will code sets as $\{\mathbf{p}\}(a, \langle b_1, ..., b_{\mathsf{n}}\rangle)$, where a is a number coding a particular constructor and $\langle b_1, ..., b_{\mathsf{n}}\rangle$ is a lists of codes for ingredients needed by the constructor itself. The following table makes evident the choices for a:

N_0, N_1, N	Π	Σ	$+$	$List$	$\bot$	$\wedge$	$\vee$	$\to$	$\exists$	$\forall$	Id
1	2	3	4	5	6	7	8	9	10	11	12

Notice that codes for small propositions must have $a > 5$.

We can now proceed to the definition of the interpretation of $\mathbf{mTT}^s$-terms.

▶ **Definition 4.** Terms in context $t[x_1, ..., x_{\mathsf{n}}]$ are interpreted as

$$\mathcal{I}(t[x_1, ..., x_{\mathsf{n}}]) = \Lambda x_1 ... \Lambda x_{\mathsf{n}}.\mathcal{I}(t)$$

where $\mathcal{I}(t)$ are terms of the extended language of $\widehat{ID_1}$ defined as follows

1. If x is a variable, then $\mathcal{I}(x) = x$;
2. $\mathcal{I}(\mathsf{emp}_0(t)) = \mathcal{I}(\mathsf{r}_0) = 0$;
3. $\mathcal{I}(\star) = 0$ and $\mathcal{I}(El_{N_1}(t, t')) = \mathcal{I}(t')$;
4. $\mathcal{I}(0) = 0$ and $\mathcal{I}(\mathsf{succ}(t)) = \{\mathbf{succ}\}(\mathcal{I}(t))$,
   $\mathcal{I}(El_N(t, t', (y, z)r)) = \{\mathbf{rec}\}(\mathcal{I}(t'), \Lambda y.\Lambda z.\mathcal{I}(r), \mathcal{I}(t))$;
5. $\mathcal{I}(\lambda y.s) = \mathcal{I}(\lambda_{\to} y.s) = \mathcal{I}(\lambda_{\forall} y.s) = \Lambda y.\mathcal{I}(s)$,
   $\mathcal{I}(\mathsf{Ap}(t, t')) = \mathcal{I}(\mathsf{Ap}_{\to}(t, t')) = \mathcal{I}(\mathsf{Ap}_{\forall}(t, t')) = \{\mathcal{I}(t)\}(\mathcal{I}(t'))$;
6. $\mathcal{I}(\langle t, t'\rangle) = \mathcal{I}(\langle t,_{\wedge} t'\rangle) = \mathcal{I}(\langle t,_{\exists} t'\rangle) = \{\mathbf{p}\}(\mathcal{I}(t), \mathcal{I}(t'))$,
   $\mathcal{I}(El_{\Sigma}(t, (y, z)r)) = \mathcal{I}(El_{\exists}(t, (y, z)r)) = \{\Lambda y.\Lambda z.\mathcal{I}(r)\}(\{\mathbf{p}_1\}(\mathcal{I}(t)), \{\mathbf{p}_2\}(\mathcal{I}(t)))$,
   $\mathcal{I}(\pi_1^{\wedge}(t)) = \{\mathbf{p}_1\}(\mathcal{I}(t))$,
   $\mathcal{I}(\pi_2^{\wedge}(t)) = \{\mathbf{p}_2\}(\mathcal{I}(t))$;
7. $\mathcal{I}(\mathsf{inl}(t)) = \mathcal{I}(\mathsf{inl}_{\vee}(t)) = \{\mathbf{p}\}(0, \mathcal{I}(t))$,
   $\mathcal{I}(\mathsf{inr}(t)) = \mathcal{I}(\mathsf{inr}_{\vee}(t)) = \{\mathbf{p}\}(1, \mathcal{I}(t))$,
   $\mathcal{I}(El_+(t, (y)s, (y)s')) = \mathcal{I}(El_{\vee}(t, (y)s, (y)s')) =$
   $\{\mathbf{ite}\}(\mathbf{p}_1(\mathcal{I}(t)), \{\Lambda y.\mathcal{I}(s)\}(\{\mathbf{p}_2\}(\mathcal{I}(t))), \{\Lambda y.\mathcal{I}(s')\}(\{\mathbf{p}_2\}(\mathcal{I}(t))))$;
8. $\mathcal{I}(\epsilon) = 0$ and $\mathcal{I}(\mathsf{cons}(t, t')) = \{\mathbf{cnc}\}(\mathcal{I}(t), \mathcal{I}(t'))$,
   $\mathcal{I}(El_{List}(t, t', (y, z, u)q)) = \{\mathbf{listrec}\}(\mathcal{I}(t'), \Lambda y.\Lambda z.\Lambda u.\mathcal{I}(q), \mathcal{I}(t))$;
9. $\mathcal{I}(\mathsf{id}(t)) = 0$,
   $\mathcal{I}(El_{\mathsf{Id}}(t, t', t'', (y)s)) = \{\Lambda y.\mathcal{I}(s)\}(\mathcal{I}(t))$;
10. $\mathcal{I}(\widehat{N_0}) = \{\mathbf{p}\}(1, 0)$, $\mathcal{I}(\widehat{N_1}) = \{\mathbf{p}\}(1, 1)$ and $\mathcal{I}(\widehat{N}) = \{\mathbf{p}\}(1, 2)$,
    $\mathcal{I}(\widehat{(\Pi y \in A)s}) = \{\mathbf{p}\}(2, (\{\mathbf{p}\}(\mathcal{I}(\widehat{A}), (\Lambda y.\mathcal{I}(s)))))$,
    $\mathcal{I}(\widehat{(\Sigma y \in A)s}) = \{\mathbf{p}\}(3, (\{\mathbf{p}\}(\mathcal{I}(\widehat{A}), (\Lambda y.\mathcal{I}(s)))))$,
    $\mathcal{I}(\widehat{t + t'}) = \{\mathbf{p}\}(4, (\{\mathbf{p}\}(\mathcal{I}(t), \mathcal{I}(t'))))$,
    $\mathcal{I}(\widehat{List}(t)) = \{\mathbf{p}\}(5, \mathcal{I}(t))$,

$$\mathcal{I}(\widehat{\bot}) = \{\mathbf{p}\}(6,0),$$
$$\mathcal{I}(t\widehat{\wedge}t') = \{\mathbf{p}\}(7,(\{\mathbf{p}\}(\mathcal{I}(t),\mathcal{I}(t')))),$$
$$\mathcal{I}(t\widehat{\vee}t') = \{\mathbf{p}\}(8,(\{\mathbf{p}\}(\mathcal{I}(t),\mathcal{I}(t')))),$$
$$\mathcal{I}(t\widehat{\rightarrow}t') = \{\mathbf{p}\}(9,(\{\mathbf{p}\}(\mathcal{I}(t),\mathcal{I}(t')))),$$
$$\mathcal{I}(\widehat{(\exists y\in A)s}) = \{\mathbf{p}\}(10,(\{\mathbf{p}\}(\mathcal{I}(\widehat{A}),(\Lambda y.\mathcal{I}(s))))),$$
$$\mathcal{I}(\widehat{(\forall y\in A)s}) = \{\mathbf{p}\}(11,(\{\mathbf{p}\}(\mathcal{I}(\widehat{A}),(\Lambda y.\mathcal{I}(s))))),$$
$$\mathcal{I}(\widehat{\mathsf{Id}}(A,t,t')) = \{\mathbf{p}\}(12,(\{\mathbf{p}\}(\mathcal{I}(\widehat{A}),(\{\mathbf{p}\}(\mathcal{I}(t),\mathcal{I}(t')))))),$$

For the sake of example let us consider the interpretation of the term in context $t[x,y,z]$ defined as $\widehat{\mathsf{Id}}(\mathsf{Id}(N,x,x),y,z)[x,y,z]$:

$$\begin{aligned}
\mathcal{I}(t)[x,y,z]) &= \Lambda x.\Lambda y.\Lambda z.\mathcal{I}(\widehat{\mathsf{Id}}(\mathsf{Id}(N,x,x),y,z))\\
&= \Lambda x.\Lambda y.\Lambda z.\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\mathcal{I}(\widehat{\mathsf{Id}(N,x,x)}),\{\mathbf{p}\}(y,z)))\\
&= \Lambda x.\Lambda y.\Lambda z.\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\mathcal{I}(\widehat{\mathsf{Id}}(N,x,x)),\{\mathbf{p}\}(y,z)))\\
&= \Lambda x.\Lambda y.\Lambda z.\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\mathcal{I}(\widehat{N}),\{\mathbf{p}\}(x,x))),\ \{\mathbf{p}\}(y,z)))\\
&= \Lambda x.\Lambda y.\Lambda z.\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\{\mathbf{p}\}(12,\ \{\mathbf{p}\}(\{\mathbf{p}\}(1,2),\{\mathbf{p}\}(x,x))),\ \{\mathbf{p}\}(y,z))).
\end{aligned}$$

We say that an interpretation of a term in context $t[\underline{x}]$ is well defined if $\mathcal{I}(t[\underline{x}])\downarrow$ is provable in $\widehat{ID_1}$. Notice that the interpretations of terms in non-empty contexts are always well defined.

Notice moreover that in $\widehat{ID_1}$

1. $\mathcal{I}(El_{N_1}(\star,t')) \simeq \mathcal{I}(t')$;
2. $\mathcal{I}(El_N(0,t,(y,z)s)) \simeq \mathcal{I}(t)$;
3. $\mathcal{I}(El_N(\mathsf{succ}(t'),t,(y,z)s)) \simeq \mathcal{I}(s)[\mathcal{I}(t')/y,\mathcal{I}(El_N(t',t,(y,z)s))/z]$
4. $\mathcal{I}(\mathsf{Ap}(\lambda y.s,t)) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
5. $\mathcal{I}(\mathsf{Ap}_{\rightarrow}(\lambda_{\rightarrow}y.s,t)) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
6. $\mathcal{I}(\mathsf{Ap}_{\forall}(\lambda_{\forall}y.s,t)) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
7. $\mathcal{I}(El_{\Sigma}(\langle t,t'\rangle,(y,z)r)) \simeq \mathcal{I}(r)[\mathcal{I}(t)/y,\mathcal{I}(t')/z]$;
8. $\mathcal{I}(El_{\exists}(\langle t,_{\exists}\, t'\rangle,(y,z)r)) \simeq \mathcal{I}(r)[\mathcal{I}(t)/y,\mathcal{I}(t')/z]$;
9. $\mathcal{I}(\pi_1^{\wedge}(\langle t,_{\wedge}\, t'\rangle)) \simeq \mathcal{I}(t)$;
10. $\mathcal{I}(\pi_2^{\wedge}(\langle t,_{\wedge}\, t'\rangle)) \simeq \mathcal{I}(t')$;
11. $\mathcal{I}(El_{+}(\mathsf{inl}(t),(y)s,(y)s')) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
12. $\mathcal{I}(El_{+}(\mathsf{inr}(t),(y)s,(y)s')) \simeq \mathcal{I}(s')[\mathcal{I}(t)/y]$;
13. $\mathcal{I}(El_{\vee}(\mathsf{inl}_{\vee}(t),(y)s,(y)s')) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
14. $\mathcal{I}(El_{\vee}(\mathsf{inr}_{\vee}(t),(y)s,(y)s')) \simeq \mathcal{I}(s')[\mathcal{I}(t)/y]$;
15. $\mathcal{I}(El_{\mathsf{Id}}(t,t,\mathsf{id}(t),(y)s)) \simeq \mathcal{I}(s)[\mathcal{I}(t)/y]$;
16. $\mathcal{I}(El_{List}(\epsilon,t',(y,z,u)q)) \simeq \mathcal{I}(t')$;
17. $\mathcal{I}(El_{List}(\mathsf{cons}(t,t''),t',(y,z,u)q)) \simeq \mathcal{I}(q)[\mathcal{I}(t)/y,\mathcal{I}(t'')/z,\mathcal{I}(El_{List}(t,t',(y,z,u)q))/u]$.

The interpretation of sets

Here we define the interpretation of sets in $\mathbf{mTT}^s$ with the exception of those obtained using $\tau(p)$ for some term p. Every such a set is interpreted as a definable quotient of a definable class of $\widehat{ID_1}$ (and actually of HA). This means that every set A is interpreted as a pair

$$\mathcal{I}(A) = (\,\mathcal{J}(A)\,,\,\sim_{\mathcal{I}(A)}\,)$$

where $\mathcal{J}(A)$ is a definable class of $\widehat{ID_1}$ and $\sim_{\mathcal{I}(A)}$ is a definable equivalence relation on the class $\mathcal{J}(A)$.

Since sets in **mTT** include small propositions, here we also define a realizability relation between natural numbers and propositions. Indeed it is more convenient to define the realizability interpretation of propositions by adopting an extension of usual Kleene's interpretation of intuitionistic connectives.

Note that we use the notation $\mathcal{I}(A)[s/y]$ to mean the definable class in which we substitute y with s in the membership and in the equivalence relation of $\mathcal{I}(A)$.

▶ **Definition 5.** We define in $\widehat{ID_1}$ a realizability relation $n \Vdash \phi$ between natural numbers and small propositions, by induction on the definition of small propositions ϕ, simultaneously together with the definition of the following formulas $n \varepsilon \mathcal{J}(A)$ and $n \sim_{\mathcal{I}(A)} m$ for sets A, by induction on the definition of sets (with the exception of those obtained using $\tau(p)$ for some term p), as follows:

$(\bot)$ $n \Vdash \bot$ is $\bot$;
$(\wedge)$ $n \Vdash \phi \wedge \phi'$ is $(p_1(n) \Vdash \phi) \wedge (p_2(n) \Vdash \phi')$;
$(\vee)$ $n \Vdash \phi \vee \phi'$ is $(p_1(n) = 0 \wedge p_2(n) \Vdash \phi) \vee (p_1(n) \neq 0 \wedge p_2(n) \Vdash \phi')$;
$(\rightarrow)$ $n \Vdash \phi \rightarrow \phi'$ is $\forall t\,((t \Vdash \phi) \rightarrow (\{n\}(t) \Vdash \phi'))$;
$(\exists)$ $n \Vdash (\exists x \in A)\,\psi$ is $p_1(n)\,\varepsilon\,\mathcal{J}(A) \wedge (p_2(n) \Vdash \psi)[p_1(n)/x]$;
$(\forall)$ $n \Vdash (\forall x \in A)\,\psi$ is $\forall x\,(x\,\varepsilon\,\mathcal{J}(A) \rightarrow (\{n\}(x) \Vdash \psi))$;
(Id) $n \Vdash \mathsf{Id}(A, t, s)$ is $\mathcal{I}(t) \sim_{\mathcal{I}(A)} \mathcal{I}(s)$;
$(\boldsymbol{N_0})$ $n\,\varepsilon\,\mathcal{J}(N_0)$ is $\bot$ and
$n \sim_{\mathcal{I}(N_0)} m$ is $\bot$;
$(\boldsymbol{N_1})$ $n\,\varepsilon\,\mathcal{J}(N_1)$ is $n = 0$ and
$n \sim_{\mathcal{I}(N_1)} m$ is $n = 0 \wedge n = m$;
$(\boldsymbol{N})$ $n\,\varepsilon\,\mathcal{J}(N)$ is $n = n$ and
$n \sim_{\mathcal{I}(N)} m$ is $n = m$;
$(\boldsymbol{\Pi})$ $n\,\varepsilon\,\mathcal{J}((\Pi x \in A)\,B)$ is
$\forall x\,(x\,\varepsilon\,\mathcal{J}(A) \rightarrow \{n\}(x) \in \mathcal{J}(B)) \wedge \forall x \forall y\,(x \sim_{\mathcal{I}(A)} y \rightarrow \{n\}(x) \sim_{\mathcal{I}(B)} \{n\}(y))$[6] and
$n \sim_{\mathcal{I}((\Pi x \in A)\,B)} m$ is
$n\,\varepsilon\,\mathcal{J}((\Pi x \in A)B) \wedge m\,\varepsilon\,\mathcal{J}((\Pi x \in A)B) \wedge \forall x\,(x\,\varepsilon\,\mathcal{J}(A) \rightarrow \{n\}(x) \sim_{\mathcal{I}(B)} \{m\}(x))$;
$(\boldsymbol{\Sigma})$ $n\,\varepsilon\,\mathcal{J}((\Sigma x \in A)\,B)$ is $p_1(n)\,\varepsilon\,\mathcal{J}(A) \wedge \forall x\,(x \sim_{\mathcal{I}(\mathsf{A})} p_1(n) \rightarrow p_2(n)\,\varepsilon\,\mathcal{J}(B))$ and
$n \sim_{\mathcal{I}((\Sigma x \in A)\,B)} m$ is the conjunction of $n\,\varepsilon\,\mathcal{J}((\Sigma x \in A)B) \wedge m\,\varepsilon\,\mathcal{J}((\Sigma x \in A)B)$ and
$p_1(n) \sim_{\mathcal{I}(A)} p_1(m) \wedge \forall x\,(x \sim_{\mathcal{I}(A)} p_1(n) \rightarrow p_2(n) \sim_{\mathcal{I}(B)} p_2(m))$;
$(+)$ $n\,\varepsilon\,\mathcal{J}(A + A')$ is $(p_1(n) = 0 \wedge p_2(n)\,\varepsilon\,\mathcal{J}(A)) \vee (p_1(n) = 1 \wedge p_2(n)\,\varepsilon\,\mathcal{J}(A'))$ and
$n \sim_{\mathcal{I}(A+A')} m$ is the conjunction of $n\,\varepsilon\,\mathcal{J}(A + A') \wedge m\,\varepsilon\,\mathcal{J}(A + A') \wedge p_1(n) = p_1(m)$ and
$(p_1(n) = 0 \wedge p_2(n) \sim_{\mathcal{I}(A)} p_2(m)) \vee (p_1(n) = 1 \wedge p_2(n) \sim_{\mathcal{I}(A')} p_2(m))$;
$(\boldsymbol{List})$ $n\,\varepsilon\,\mathcal{J}(List(A))$ is $\forall j\,(j < lh(n) \rightarrow (n)_j\,\varepsilon\,\mathcal{J}(A))$ and
$n \sim_{\mathcal{I}(List(A))} m$ is the conjunction of $n\,\varepsilon\,\mathcal{J}(List(A)) \wedge m\,\varepsilon\,\mathcal{J}(List(A))$ and
$lh(n) = lh(m) \wedge \forall j\,(j < lh(n) \rightarrow (n)_j \sim_{\mathcal{I}(A)} (m)_j)$;
$(\boldsymbol{\psi})$ $n\,\varepsilon\,\mathcal{J}(\psi)$ is $n \Vdash \psi$ and
$n \sim_{\mathcal{I}(\psi)} m$ is $n\,\varepsilon\,\mathcal{J}(\psi) \wedge m\,\varepsilon\,\mathcal{J}(\psi)$ (i. e. *proof-irrelevance*).

▶ Remark. We can notice some preliminary properties of this realizability interpretation:

1. for every set A we have that $\sim_{\mathcal{I}(A)}$ is really a definable equivalence relation on the definable class $\mathcal{J}(A)$, in fact

[6] Note that the variable x may be in $\mathcal{I}(B)$ here and in the following definition for Π and Σ sets, as it comes from the definition of the untyped syntax.

$$n \,\varepsilon\, \mathcal{J}(A) \vdash_{\widehat{ID_1}} n \sim_{\mathcal{I}(A)} n$$
$$n \sim_{\mathcal{I}(A)} m \vdash_{\widehat{ID_1}} m \sim_{\mathcal{I}(A)} n$$
$$n \sim_{\mathcal{I}(A)} m \wedge m \sim_{\mathcal{I}(A)} l \vdash_{\widehat{ID_1}} n \sim_{\mathcal{I}(A)} l$$

2. for every set A we have that

$$n \sim_{\mathcal{I}(A)} m \vdash_{\widehat{ID_1}} n \,\varepsilon\, \mathcal{J}(A) \wedge m \,\varepsilon\, \mathcal{J}(A)$$

3. if *numerical* sets are defined according to the following conditions
   a. N_0, N_1 and N are numerical sets;
   b. if A and B are numerical sets, then $(\Sigma x \in A)\, B$, $A + B$ and $List(A)$ (if they are well defined) are numerical sets,

   then the equality of the interpretation of numerical sets is numerical, which means that

$$n \sim_{\mathcal{I}(A)} m \vdash_{\widehat{ID_1}} n = m$$

4. for all small propositions ψ, the equivalence relation $\sim_{\mathcal{I}(\psi)}$ is trivial (i. e. all pairs of elements of $\mathcal{I}(\psi)$ are equivalent). This means that uniqueness of propositional proofs, called *proof-irrelevance*, is imposed.

The encoding of all mTT^s-sets

In the previous sections we have seen the interpretation of **mTT**^{s}-sets which include small propositions. It remains to define the interpretation of proper collections, including that of sets, small propositions and small propositional functions on a set.

The interpretation of the collection of small propositions Set in $\widehat{ID_1}$ is the most difficult point and to define it we mimick the technique adopted by Beeson [2] to interpret Martin-Löf's universe via a fix point of some arithmetical operator with positive parameters. Hence, it is to define the interpretation of Set, and in turn of the collection of small propositions $\mathsf{prop_s}$ and of small propositional functions $A \to \mathsf{prop_s}$ on a set A, that we need to employ the full power of $\widehat{ID_1}$ with fix points.

The idea is to define a $\widehat{ID_1}$-formula which defines codes of sets with their interpretation *as a fix point*. It appears necessary to define a formula called $\mathbf{Set}(n)$ expressing that n is a code of an **mTT**^{s}-set together with its realizability interpretation in $\widehat{ID_1}$. As in Beeson's semantics, to define the formula $\mathbf{Set}(n)$ of set codes with their arithmetical interpretation in $\widehat{ID_1}$ we need to encode membership and equality of sets: $t \,\bar{\varepsilon}\, n$ and $t \equiv_n s$. In turn in order to define them, we need to represent the notion of *a family of sets* used to interpret an **mTT**^{s}-dependent set. *A family of sets* coded by m on a set coded by n could be described by the formula

$$\mathbf{Set}(n) \wedge \forall t\,(t \,\bar{\varepsilon}\, n \to \mathbf{Set}(\{m\}(t))) \wedge$$
$$\forall t \forall s\,(t \equiv_n s \to (\forall j\,(j \,\bar{\varepsilon}\, \{m\}(t) \leftrightarrow j \,\bar{\varepsilon}\, \{m\}(s)) \wedge \forall j \forall k\,(j \equiv_{\{m\}(t)} k \leftrightarrow j \equiv_{\{m\}(s)} k))).$$

But in this formula not all occurrences of $t \,\bar{\varepsilon}\, n$ and $t \equiv_n s$ are positive. However it is classically equivalent to the conjunction of the formula $\mathbf{Set}(n) \wedge \forall t\,(\neg t \,\bar{\varepsilon}\, n \vee \mathbf{Set}(\{m\}(t)))$ and the formula $\forall t \forall s\,(\neg t \equiv_n s \vee (P_1 \wedge P_2))$ where P_1 is

$$\forall j\,((\neg j \,\bar{\varepsilon}\, \{m\}(t) \vee j \,\bar{\varepsilon}\, \{m\}(s)) \wedge (\neg j \,\bar{\varepsilon}\, \{m\}(s) \vee j \,\bar{\varepsilon}\, \{m\}(t)))$$

and P_2 is

$$\forall j \forall k\, ((\neg j \equiv_{\{m\}(t)} k \vee j \equiv_{\{m\}(s)} k) \wedge (\neg j \equiv_{\{m\}(s)} k \vee j \equiv_{\{m\}(t)} k))$$

simply substituting all the instances of the schema $a \to b$ with the classically equivalent $\neg a \vee b$. Now the trick consists in defining some predicates $t \notin n$ and $t \not\equiv_n s$ mimicking the negations of $t \,\overline{\in}\, n$ and $t \equiv_n s$ as fix point predicates, too, in order to get a a **positive** arithmetical operator. Note that the use of a **classical** arithmetic theory with fix points seems unavoidable to be able to interpret the collection of sets via a positive arithmetical operator. From now on we write

$$\mathbf{Fam}(m,n) \;\equiv\; \mathbf{Set}(n) \wedge \forall t\, (t \notin n \vee \mathbf{Set}(\{m\}(t))) \wedge \forall t \forall s\, (t \not\equiv_n s \vee (P_1' \wedge P_2'))$$

where P_1' and P_2' are obtained from P_1 and P_2 by substituting negated instances of membership and of equality predicates with their mentioned primitive negated versions

$$P_1' \equiv \forall j\, ((j \notin \{m\}(t) \vee j \,\overline{\in}\, \{m\}(s)) \wedge (j \notin \{m\}(s) \vee j \,\overline{\in}\, \{m\}(t)))$$

$$P_2' \equiv \forall j \forall k\, ((j \not\equiv_{\{m\}(t)} k \vee j \equiv_{\{m\}(s)} k) \wedge (j \not\equiv_{\{m\}(s)} k \vee j \equiv_{\{m\}(t)} k)).$$

In order to define the positive clauses for the codes of sets we must introduce some notations. In this way we transform the clauses for realizability for sets automatically in the clauses needed to define the fix points $\mathbf{Set}(n)$, $t \overline{\in} n$, $t \notin n$, $t \equiv_n s$ and $t \not\equiv_n s$.

First of all, we define a function $[\,]$ which assigns a value to a set according to the table in Section 3.2 as follows.

1. If σ is one of the symbols A, A', B, ϕ, ϕ', ψ, t, s, then $[\sigma]$ is a, a', $\{b\}(x)$, c, c', $\{d\}(x)$, e, f respectively;
2. If σ is N_0, N_1, N, $(\Pi x \in A)\, B$, $(\Sigma x \in A)\, B$, $A + A'$, $List(A)$ then $[\sigma]$ is $\langle 1,0\rangle$, $\langle 1,1\rangle$, $\langle 1,2\rangle$, $\langle 2,\langle a,b\rangle\rangle$, $\langle 3,\langle a,b\rangle\rangle$, $\langle 4,\langle a,a'\rangle\rangle$, $\langle 5,a\rangle$ respectively;
3. If σ is $\bot$, $\phi \wedge \phi'$, $\phi \vee \phi'$, $\phi \to \phi'$, $(\exists x \in A)\, \psi$, $(\forall x \in A)\, \psi$, $\mathsf{Id}(A,t,s)$ then $[\sigma]$ is $\langle 6,0\rangle$, $\langle 7,\langle c,c'\rangle\rangle$, $\langle 8,\langle c,c'\rangle\rangle$, $\langle 9,\langle c,c'\rangle\rangle$, $\langle 10,\langle a,d\rangle\rangle$, $\langle 11,\langle a,d\rangle\rangle$, $\langle 12,\langle a,\langle e,f\rangle\rangle\rangle$ respectively.

We denote by $[\,]^{-1}$ the inverse function of $[\,]$. Now, all clauses in the realizability interpretation of sets are defined using formulas which are obtained starting from arithmetical formulas or primitive formulas with ε or $\sim$, by using connectives, first order quantifiers or explicit instances of substitution in x. For such formulas φ we define φ^+ as follows:

1. If φ is arithmetical, then φ^+ is defined as φ itself. If φ is a primitive formulas with ε or $\sim$ we will transform $\varepsilon\, \mathcal{J}(\sigma)$ and $\sim_{\mathcal{I}(\sigma)}$ in $\overline{\varepsilon}\,[\sigma]$ and $\equiv_{[\sigma]}$ respectively, in order to obtain φ^+;
2. $(\varphi[\alpha/x])^+$ is $\varphi^+[\alpha/x]$;
3. $(\varphi \wedge \varphi')^+$ is $\varphi^+ \wedge \varphi'^+$;
4. $(\varphi \vee \varphi')^+$ is $\varphi^+ \vee \varphi'^+$;
5. $(\varphi \to \varphi')^+$ is $\overline{\varphi^+} \vee \varphi'^+$;
6. $(\forall u\, \varphi)^+$ is $\forall u\, \varphi^+$ for every variable u;
7. $(\exists u\, \varphi)^+$ is $\exists u\, \varphi^+$ for every variable u;

where $\overline{\varphi}$ is defined by the following clauses:

1. If φ is an arithmetical formula $\overline{\varphi}$ is $\neg\varphi$;
2. If φ is a relation between two terms through $\overline{\varepsilon}$, $\notin$, $\equiv$ or $\not\equiv$, then $\overline{\varphi}$ is obtained by transforming them in $\notin$, $\overline{\varepsilon}$, $\not\equiv$ or $\equiv$ respectively;
3. $\overline{\varphi \wedge \varphi'}$ is $\overline{\varphi} \vee \overline{\varphi'}$;
4. $\overline{\varphi \vee \varphi'}$ is $\overline{\varphi} \wedge \overline{\varphi'}$;
5. $\overline{\forall u\, \varphi}$ is $\exists u\, \overline{\varphi}$ for every variable u;
6. $\overline{\exists u\, \varphi}$ is $\forall u\, \overline{\varphi}$ for every variable u;

We can now define the positive clauses we needed. For τ equal to $\langle 1,0\rangle$, $\langle 1,1\rangle$, $\langle 1,2\rangle$, $\langle 2,\langle a,b\rangle\rangle$, $\langle 3,\langle a,b\rangle\rangle$, $\langle 4,\langle a,a'\rangle\rangle$, $\langle 5,a\rangle$, $\langle 6,0\rangle$, $\langle 7,\langle c,c'\rangle\rangle$, $\langle 8,\langle c,c'\rangle\rangle$, $\langle 9,\langle c,c'\rangle\rangle$, $\langle 10,\langle a,d\rangle\rangle$, $\langle 11,\langle a,d\rangle\rangle$, $\langle 12,\langle a,\langle e,f\rangle\rangle\rangle$ we have the following clauses[7]:

1. $\mathbf{Set}(\tau)$ if $\mathsf{Cond}(\tau)$;
2. $n\,\bar{\varepsilon}\,\tau$ if $\mathsf{Cond}(\tau) \wedge \overline{(n\,\varepsilon\,\mathcal{J}([\tau]^{-1}))^{+}}$;
3. $n\,\bar{\notin}\,\tau$ if $\mathsf{Cond}(\tau) \wedge \overline{(n\,\varepsilon\,\mathcal{J}([\tau]^{-1}))^{+}}$;
4. $n \equiv_\tau m$ if $\mathsf{Cond}(\tau) \wedge \overline{(n \sim_{\mathcal{I}([\tau]^{-1})} m)^{+}}$;
5. $n \not\equiv_\tau m$ if $\mathsf{Cond}(\tau) \wedge \overline{(n \sim_{\mathcal{I}([\tau]^{-1})} m)^{+}}$;

where $\mathsf{Cond}(\tau)$ is

1. $\top$ if τ has first component 1 or 6;
2. $\mathbf{Fam}(b,a)$ if τ has first component 2 or 3;
3. $\mathbf{Set}(a) \wedge \mathbf{Set}(a')$ if τ has first component 4;
4. $\mathbf{Set}(a)$ if τ has first component 5;
5. $\mathbf{Set}(c) \wedge \mathbf{Set}(c') \wedge \pi_1(c) > 5 \wedge \pi_1(c') > 5$ if τ has first component 7, 8 or 9;
6. $\mathbf{Fam}(d,a) \wedge \forall x\,(x\,\bar{\notin}\,a \vee \pi_1(\{d\}(x)) > 5)$ if τ has first component 10, 11;
7. $\mathbf{Set}(a) \wedge e\,\bar{\varepsilon}\,a \wedge f\,\bar{\varepsilon}\,a$ if τ has first component 12.

By sake of example we present here the clauses for codes of Π-sets.

$\mathbf{Set}(\langle 2,\langle a,b\rangle\rangle)$ if $\mathbf{Fam}(b,a)$;
$n\,\bar{\varepsilon}\,\langle 2,\langle a,b\rangle\rangle$ if

$$\mathbf{Fam}(b,a) \wedge \forall x\,(x\,\bar{\notin}\,a \vee \{n\}(x)\,\bar{\varepsilon}\,\{b\}(x)) \wedge \forall x\forall y\,(x \not\equiv_a y \vee \{n\}(x) \equiv_{\{b\}(x)} \{n\}(y));$$

$n\,\bar{\notin}\,\langle 2,\langle a,b\rangle\rangle$ if

$$\mathbf{Fam}(b,a) \wedge (\exists x\,(x\,\bar{\varepsilon}\,a \wedge \{n\}(x)\,\bar{\notin}\,\{b\}(x)) \vee \exists x\exists y\,(x \equiv_a y \wedge \{n\}(x) \not\equiv_{\{b\}(x)} \{n\}(y)));$$

$n \equiv_{\langle 2,\langle a,b\rangle\rangle} m$ if

$$\mathbf{Fam}(b,a) \wedge n\,\bar{\varepsilon}\,\langle 2,\langle a,b\rangle\rangle \wedge m\,\bar{\varepsilon}\,\langle 2,\langle a,b\rangle\rangle \wedge \forall x\,(x\,\bar{\notin}\,a \vee \{n\}(x) \equiv_{\{b\}(x)} \{m\}(x));$$

$n \not\equiv_{\langle 2,\langle a,b\rangle\rangle} m$ if

$$\mathbf{Fam}(b,a) \wedge (n\,\bar{\notin}\,\langle 2,\langle a,b\rangle\rangle \vee m\,\bar{\notin}\,\langle 2,\langle a,b\rangle\rangle \vee \exists x\,(x\,\bar{\varepsilon}\,a \wedge \{n\}(x) \not\equiv_{\{b\}(x)} \{m\}(x))$$

The formulas $\mathbf{Set}(n)$, $t\bar{\varepsilon}n$, $t\,\bar{\notin}\,n$, $t \equiv_n s$ and $t \not\equiv_n s$ are components of a predicate $P_\theta(n)$ defined in $\widehat{ID_1}$ as a fix point of an operator $\theta(n,X)$ defined by glueing together the clauses expressing the code of each $\mathbf{mTT}^s$-set-constructor with its interpretation in $\widehat{ID_1}$.

The interpretation of collections

Here we extend the realizability relation, membership and equality in definition 5 in order to interpret collections, propositions and the decoding operators.

▶ **Definition 6.** $n \Vdash \phi$ between natural numbers and $\mathbf{mTT}^s$-propositions and formulas $n\varepsilon\mathcal{J}(D)$ and $n \sim_{\mathcal{I}(D)} m$ for collections D are defined by including all clauses in definition 5 plus the following:

[7] By $n\varepsilon\mathcal{J}([\tau]^{-1})$ and $n \sim_{\mathcal{I}([\tau]^{-1})} m$ we mean the right-hand side of the respective clause in the realizability interpretation of sets, taking into account that for small propositions membership coincides with the realizability relation.

1. $n \Vdash \tau(p)$ and $n\,\varepsilon\,\mathcal{J}(\tau(p))$ are both given by $n\,\overline{\varepsilon}\,\mathcal{I}(p)$:
   $n \sim_{\mathcal{I}(\tau(p))} m$ is $n\,\varepsilon\,\mathcal{J}(\tau(p)) \wedge m\,\varepsilon\,\mathcal{J}(\tau(p))$.
2. The realizability relation $n \Vdash \eta$ for propositions is completely analogous to the realizability relation for small propositions and the interpretation of propositions is given by the class of realizers equipped with the trivial equivalence relation.
3. Σ-collections are interpreted exactly in the same way as Σ-sets.
4. $n\varepsilon\mathcal{J}(\mathsf{Set})$ is $\mathbf{Set}(n) \wedge \forall t\,(t\,\overline{\varepsilon}\,n \leftrightarrow \neg t\,\overline{\varepsilon}\,n) \wedge \forall t\forall s\,(t \equiv_n s \leftrightarrow \neg t \not\equiv_n s)$. This is because $\overline{\varepsilon}$ and $\not\equiv$, which are defined by fix point, don't behave necessarily as negations of $\overline{\varepsilon}$ and $\equiv$ and hence we need to add $\forall t\,(t\,\overline{\varepsilon}\,n \leftrightarrow \neg t\,\overline{\varepsilon}\,n)$ and $\forall t\forall s\,(t \equiv_n s \leftrightarrow \neg t \not\equiv_n s)$.
   The interpretation of $n \sim_{\mathcal{I}(\mathsf{Set})} m$ is

$$n\,\varepsilon\,\mathcal{J}(\mathsf{Set}) \wedge m\,\varepsilon\,\mathcal{J}(\mathsf{Set}) \wedge \forall t\,(t\,\overline{\varepsilon}\,n \leftrightarrow t\,\overline{\varepsilon}\,m) \wedge \forall t\forall s\,(t \equiv_n s \leftrightarrow t \equiv_m s)\,.$$

5. $n\,\varepsilon\,\mathcal{J}(\mathsf{prop_s})$ is $n\,\varepsilon\,\mathcal{J}(\mathsf{Set}) \wedge \pi_1(n) > 5 \wedge \forall t\forall s\,(t\,\overline{\varepsilon}\,n \wedge s\,\overline{\varepsilon}\,n \leftrightarrow t \equiv_n s)$ (recall that small propositions are encoded with $\pi_1(n) > 5$ and enjoy the proof-irrelevance).
   The interpretation of $n \sim_{\mathcal{I}(\mathsf{prop_s})} m$ is $n\,\varepsilon\,\mathcal{J}(\mathsf{prop_s}) \wedge m\,\varepsilon\,\mathcal{J}(\mathsf{prop_s}) \wedge \forall t\,(t\,\overline{\varepsilon}\,n \leftrightarrow t\,\overline{\varepsilon}\,m)$;
6. $n\,\varepsilon\,\mathcal{J}(A \to \mathsf{prop_s})$ is $\forall t\forall s\,(t \sim_{\mathcal{I}(A)} s \to \{n\}(t) \sim_{\mathcal{I}(\mathsf{prop_s})} \{n\}(s))$
   and $n \sim_{\mathcal{I}(A\to\mathsf{prop_s})} m$ is
   $n\,\varepsilon\,\mathcal{J}(A \to \mathsf{prop_s}) \wedge m\,\varepsilon\,\mathcal{J}(A \to \mathsf{prop_s}) \wedge \forall t\,(t\,\varepsilon\,\mathcal{J}(A) \to \{n\}(t) \sim_{\mathcal{I}\,.(\mathsf{prop_s})} \{m\}(t))$

▶ Remark. Notice that the properties of sets and small propositions stated after definition 5 hold also for collections and propositions respectively.

The interpretation of judgements

We now need to say how judgements are interpreted in our realizability model. First of all, if $\mathcal{A} = (A, \simeq_{\mathcal{A}})$ and $\mathcal{B} = (B, \simeq_{\mathcal{B}})$ are definable classes of $\widehat{ID_1}$ equipped with a definable equivalence relation, then we denote with $\mathcal{A} \doteq \mathcal{B}$ the formula $\forall t\forall s\,(t \simeq_{\mathcal{A}} s \leftrightarrow t \simeq_{\mathcal{B}} s)$.

The judgements of $\mathbf{mTT}^s$ are interpreted as follows:

1. If $type \in \{set, col, prop_s, prop\}$, the interpretation of $A\ type$ is $\mathcal{I}(A) \doteq \mathcal{I}(A)$.
2. If $type \in \{set, col, prop_s, prop\}$, the interpretation of $A = B\ type$ is $\mathcal{I}(A) \doteq \mathcal{I}(B)$.
3. The judgement $t \in A$ is interpreted as $\mathcal{I}(t)$.
4. The judgement $t = s \in A$ is interpreted as $\mathcal{I}(t) \sim_{\mathcal{I}(A)} \mathcal{I}(s)$.
5. If $type \in \{set, col, prop_s, prop\}$, the interpretation of $A\ type\ [x_1 \in A_1, ..., x_\mathsf{n} \in A_\mathsf{n}]$ is

$$\forall x_1\forall y_1 ...\forall x_\mathsf{n}\forall y_\mathsf{n}\,(x_1 \sim_{\mathcal{I}(A_1)} y_1 \wedge ... \wedge x_\mathsf{n} \sim_{\mathcal{I}(A_\mathsf{n})} y_\mathsf{n} \to \mathcal{I}(A) \doteq \mathcal{I}(A)\,[y_1/x_1, ..., y_\mathsf{n}/x_\mathsf{n}])\ ^{8}\,.$$

6. If $type \in \{set, col, prop_s, prop\}$, the interpretation of $A = B\ type\ [x_1 \in A_1, ..., x_\mathsf{n} \in A_\mathsf{n}]$ is

$$\forall x_1 ...\forall x_\mathsf{n}\,(x_1\,\varepsilon\,\mathcal{J}(A_1) \wedge ... \wedge x_\mathsf{n}\,\varepsilon\,\mathcal{J}(A_\mathsf{n}) \to \mathcal{I}(A) \doteq \mathcal{I}(B))\,.$$

7. The judgement $t \in A[x_1 \in A_1, ..., x_\mathsf{n} \in A_\mathsf{n}]$ is interpreted as

$$\forall x_1\forall y_1 ...\forall x_\mathsf{n}\forall y_\mathsf{n}\,(x_1 \sim_{\mathcal{I}(A_1)} y_1 \wedge ... \wedge x_\mathsf{n} \sim_{\mathcal{I}(A_\mathsf{n})} y_\mathsf{n} \to \mathcal{I}(t) \sim_{\mathcal{I}(A)} \mathcal{I}(t)\,[y_1/x_1, .., y_\mathsf{n}/x_\mathsf{n}])\,.$$

8. The judgement $t = s \in A[x_1 \in A_1, ..., x_\mathsf{n} \in A_\mathsf{n}]$ is interpreted as

$$\forall x_1 ...\forall x_\mathsf{n}\,(x_1\,\varepsilon\,\mathcal{J}(A_1) \wedge ... \wedge x_\mathsf{n}\,\varepsilon\,\mathcal{J}(A_\mathsf{n}) \to \mathcal{I}(t) \sim_{\mathcal{I}(A)} \mathcal{I}(s))\,.$$

3.3 The validity theorem

A judgement J in the language of $\mathbf{mTT}^s$ is validated by the realizability model $(\mathcal{R} \vDash J)$ if $\widehat{ID_1} \vdash \mathcal{I}(J)$, where $\mathcal{I}(J)$ is the interpretation of J according to the previous section. We say that a proposition ϕ is validated by the model $(\mathcal{R} \vDash \phi)$, if its interpretation can be proven to be inhabited, which means that

$\widehat{ID_1} \vdash \exists r(r \varepsilon \mathcal{J}(\phi))$ which is equivalent to $\widehat{ID_1} \vdash \exists r(r \Vdash \phi)$.

In order to prove how substitution is interpreted in a easy way, it is convenient to modify the presentation of $\mathbf{mTT}^s$-rules, into an equivalent system (still denoted by $\mathbf{mTT}^s$), where we supply the information that the members in a type equality judgement are types, and members of term equality judgements are typed terms as follows with the warning of avoiding repetitions of same judgements: for $type \in \{set, col, prop_s, prop\}$

any rule $\dfrac{J_1 ... J_n}{A = B \; type \; [\Gamma]}$ is changed to $\dfrac{J_1 ... J_n, \; A \; type \; [\Gamma] \; , \; B \; type \; [\Gamma]}{A = B \; type \; [\Gamma]}$

any rule $\dfrac{J_1 ... J_n}{b \in B \; [\Gamma]}$ is changed to $\dfrac{J_1 ... J_n, \; B \; type \; [\Gamma]}{b \in B \; [\Gamma]}$

any rule $\dfrac{J_1 ... J_n}{a = b \in A \; [\Gamma]}$ is changed to $\dfrac{J_1 ... J_n, \; a \in A \; type \; [\Gamma] \; , \; b \in A \; type \; [\Gamma]}{a = b \in A \; type \; [\Gamma]}$

the substitution rule subT) and sub) in [9] are changed to

$$\text{subT}_m) \; \frac{\begin{array}{l} C(x_1, \ldots, x_n) \; type \; [\, x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1}) \,] \\ a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}) \;\; b_1 \in A_1, \ldots, b_n \in A_n(b_1, \ldots, b_{n-1}) \\ a_1 = b_1 \in A_1 \ldots a_n = b_n \in A_n(a_1, \ldots, a_{n-1}) \end{array}}{C(a_1, \ldots, a_n) = C(b_1, \ldots, b_n) \; type}$$

$$\text{sub}_m) \; \frac{\begin{array}{l} c(x_1, \ldots, x_n) \in C(x_1, \ldots, x_n) \;\; [\, x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1}) \,] \\ C(x_1, \ldots, x_n) \; type \; [\, x_1 \in A_1, \ldots, x_n \in A_n(x_1, \ldots, x_{n-1}) \,] \\ a_1 \in A_1, \ldots, a_n \in A_n(a_1, \ldots, a_{n-1}) \;\; b_1 \in A_1, \ldots, b_n \in A_n(b_1, \ldots, b_{n-1}) \\ a_1 = b_1 \in A_1 \ldots a_n = b_n \in A_n(a_1, \ldots, a_{n-1}) \end{array}}{c(a_1, \ldots, a_n) = c(b_1, \ldots, b_n) \in C(a_1, \ldots, a_n)}$$

the formation rules F-Σ), F-$\exists$) and F-$\forall$) are changed to

$$\text{F-}\Sigma) \; \frac{\begin{array}{l} B \; col \\ C(x) \;\; col \; [x \in B] \end{array}}{\Sigma_{x \in B} C(x) \; col} \qquad \text{F-}\exists) \; \frac{\begin{array}{l} B \; col \\ C(x) \; prop \;\; [x \in B] \end{array}}{\exists_{x \in B} C(x) \; prop} \qquad \text{F-}\forall) \; \frac{\begin{array}{l} B \; col \\ C(x) \; prop \; [x \in B] \end{array}}{\forall_{x \in B} C(x) \; prop}$$

the elimination rules E-Π) and E-$\forall$) are changed to

$$\text{E-}\Pi_m) \; \frac{\begin{array}{ll} C(x) \; set \, [x \in B] & C(b) \; set \\ b \in B & f \in \Pi_{x \in B} C(x) \end{array}}{\mathsf{Ap}(f, b) \in C(b)} \qquad \text{E-}\forall_m) \; \frac{\begin{array}{ll} C(x) \; prop \, [x \in B] & C(b) \; prop \\ b \in B & f \in \forall_{x \in B} C(x) \end{array}}{\mathsf{Ap}_\forall(f, b) \in C(b)}$$

Note that each $\mathbf{mTT}^s$-type is a collection and therefore in deriving a typed term $b \in B$ under a context the addition of the information that the type B is a collection in the premise is certainly valid.

▶ **Theorem 7** (Validity theorem). *For every judgement J in the language of $\boldsymbol{mTT}^s$, if J can be proven in $\boldsymbol{mTT}^s$ $(\boldsymbol{mTT}^s \vdash J)$, then J is validated by the model $(\mathcal{R} \vDash J)$.*

Proof. In order to prove the validity theorem it is necessary to prove by induction on the height of the proof tree in $\mathbf{mTT}^s$ these three facts at the same time:

1. For every judgement J in the language of $\mathbf{mTT}^s$, if $\mathbf{mTT}^s \vdash J$ then $\mathcal{R} \vDash J$.
2. (Substitution) If $\mathbf{mTT}^s \vdash C\, type\, [x_1 \in A_1, ..., x_{\mathsf{n}} \in A_{\mathsf{n}}]$ for $type \in \{set, col, prop_s, prop\}$ for all

$$\mathbf{mTT}^s \vdash a_1 \in A_1[y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}], ...,$$

$$\mathbf{mTT}^s \vdash a_{\mathsf{n}} \in A_{\mathsf{n}}[a_1/x_1, ..., a_{\mathsf{n}-1}/x_{\mathsf{n}-1}][y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}],$$

if $\mathcal{R} \vDash a_1 \in A_1[y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}]$,...,

$$\mathcal{R} \vDash a_{\mathsf{n}} \in A_{\mathsf{n}}[a_1/x_1, ..., a_{\mathsf{n}-1}/x_{\mathsf{n}-1}][y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}],$$

then

$$\widehat{ID_1} \vdash \forall y_1 ... \forall y_{\mathsf{m}}\, (y_1\, \varepsilon\, \mathcal{J}(B_1) \wedge ... \wedge y_{\mathsf{m}}\, \varepsilon\, \mathcal{J}(B_{\mathsf{m}}) \rightarrow \mathcal{I}(C)\, [\mathcal{I}(a_1)/x_1, ..., \mathcal{I}(a_{\mathsf{n}})/x_{\mathsf{n}}] \doteq \mathcal{I}(C\, [a_1/x_1, ..., a_{\mathsf{n}}/x_{\mathsf{n}}]))$$

and if $\mathbf{mTT}^s \vdash c \in C[x_1 \in A_1, ..., x_{\mathsf{n}} \in A_{\mathsf{n}}]$ for all

$$\mathbf{mTT}^s \vdash a_1 \in A_1[y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}], ...,$$

$$\mathbf{mTT}^s \vdash a_{\mathsf{n}} \in A_{\mathsf{n}}[a_1/x_1, ..., a_{\mathsf{n}-1}/x_{\mathsf{n}-1}][y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}],$$

if $\mathcal{R} \vDash a_1 \in A_1[y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}]$,...,

$$\mathcal{R} \vDash a_{\mathsf{n}} \in A_{\mathsf{n}}[a_1/x_1, ..., a_{\mathsf{n}-1}/x_{\mathsf{n}-1}][y_1 \in B_1, ..., y_{\mathsf{m}} \in B_{\mathsf{m}}],$$

then

$$\widehat{ID_1} \vdash \forall y_1 ... \forall y_{\mathsf{m}}\, (y_1\, \varepsilon \mathcal{J}(B_1) \wedge ... \wedge y_{\mathsf{m}}\, \varepsilon\, \mathcal{J}(B_{\mathsf{m}}) \rightarrow \mathcal{I}(c)\, [\mathcal{I}(a_1)/x_1, ..., \mathcal{I}(a_{\mathsf{n}})/x_{\mathsf{n}}] \sim_{\mathcal{I}(C\, [a_1/x_1, ..., a_{\mathsf{n}}/x_{\mathsf{n}}])} \mathcal{I}(c\, [a_1/x_1, ..., a_{\mathsf{n}}/x_{\mathsf{n}}])).$$

3. (Coding) If $\mathbf{mTT}^s \vdash B\, set\, [x_1 \in A_1, ..., x_{\mathsf{n}} \in A_{\mathsf{n}}]$, then

$$\widehat{ID_1} \vdash \forall x_1 ... \forall x_{\mathsf{n}}\, (x_1\, \varepsilon\, \mathcal{J}(A_1) \wedge ... \wedge x_{\mathsf{n}}\, \varepsilon\, \mathcal{J}(A_{\mathsf{n}}) \rightarrow \mathbf{Set}(\mathcal{I}(\hat{B})) \wedge \forall t\, (t\, \varepsilon\, \mathcal{J}(B) \leftrightarrow t\, \bar{\varepsilon}\, \mathcal{I}(\hat{B})) \wedge \forall t\, (\neg t\, \varepsilon\, \mathcal{J}(B) \leftrightarrow t\, \bar{\not\varepsilon}\, \mathcal{I}(\hat{B})) \wedge \forall t \forall s\, (t \sim_{\mathcal{I}(B)} s \leftrightarrow t \equiv_{\mathcal{I}(\hat{B})} s) \wedge \forall t \forall s\, (\neg t \sim_{\mathcal{I}(B)} s \leftrightarrow t \not\equiv_{\mathcal{I}(\hat{B})} s)).$$

Let us show only some cases, as the techniques will be similar in the other cases. In particular we will consider validity and substitution for the rule of introduction and validity for the rule of conversion for Π-sets. The technique is similar in the other cases.

1. Suppose that we derived in $\mathbf{mTT}^s$ the judgement $\lambda y.c \in (\Pi y \in B)\, C\, [x \in A]$ by introduction rule, after having derived $c \in C\, [x \in A, y \in B]$ and $(\Pi y \in B)\, C\, set\, [x \in A]$. By inductive hypothesis on validity we can suppose that $\mathcal{R} \vDash c \in C\, [x \in A, y \in B]$. This means that in $\widehat{ID_1}$

$$\forall x \forall x' \forall y \forall y'\, (x \sim_{\mathcal{I}(A)} x' \wedge y \sim_{\mathcal{I}(B)} y' \rightarrow \mathcal{I}(c) \sim_{\mathcal{I}(C)} \mathcal{I}(c)\, [x'/x, y'/y]\,)$$

which is equivalent to

$$\forall x \forall x' (x \sim_{\mathcal{I}(A)} x' \rightarrow \forall y \forall y' (y \sim_{\mathcal{I}(B)} y' \rightarrow \{\Lambda x.\Lambda y.\mathcal{I}(c)\}(x, y) \sim_{\mathcal{I}(C)} \{\Lambda x.\Lambda y.\mathcal{I}(c)\}(x', y'))$$

Using this fact together with the fact that, by inductive hypothesis on validity (as the judgement $B\, set\, [x \in A]$ is derived with a shorter derivation), in $\widehat{ID_1}$ we have that $\forall x \forall x'\, (x \sim_{\mathcal{I}(A)} x' \rightarrow \mathcal{I}(B) \doteq \mathcal{I}(B)[x'/x])$, we obtain that

$$\widehat{ID_1} \vdash \forall x \forall x'\, (x \sim_{\mathcal{I}(A)} x' \rightarrow \mathcal{I}(\lambda y.c) \sim_{\mathcal{I}((\Pi y \in B)C)} \mathcal{I}(\lambda y.c)\, [x'/x]).$$

2. We want to prove the substitution property for a judgement $\lambda y.c \in (\Pi y \in B)\, C\,[x \in A]$ derived in **mTT** by introduction after having derived $c \in C\,[x \in A, y \in B]$ and $(\Pi y \in B)\, C\, set\,[x \in A]$ with respect to a term a for which $\mathbf{mTT} \vdash a \in A$ and $\mathcal{R} \vDash a \in A$. First of all notice that by the structure of derivations in **mTT**, $B\, set\,[x \in A]$ and (so also) $B[a/x]\, set$ can be derived in **mTT**. From this it follows that $\mathbf{mTT} \vdash a \in A[y \in B[a/x]]$ and $\mathbf{mTT} \vdash y \in B[a/x]\,[y \in B[a/x]]$. These judgements are also validated by $\mathcal{R}$ by definition of the interpretation and because $\mathcal{R} \vDash a \in A$. By inductive hypothesis on substitution applied to $c \in C\,[x \in A, y \in B]$ with respect to $a \in A[y \in B[a/x]]$ and $y \in B[a/x]\,[y \in B[a/x]]$ we obtain that in $\widehat{ID_1}$ it holds that $\forall y\,(y \varepsilon \mathcal{J}(B[a/x]) \to \mathcal{I}(c)[\mathcal{I}(a)/x] \sim_{\mathcal{I}(C[a/x])} \mathcal{I}(c[a/x]))$ and this entails that

$$\widehat{ID_1} \vdash \forall y\,(y\,\varepsilon\,\mathcal{J}(B[a/x]) \to \{\mathcal{I}(\lambda y.c)[\mathcal{I}(a)/x]\}(y) \sim_{\mathcal{I}(C[a/x])} \{\mathcal{I}(\lambda y.c[a/x])\}(y)).$$

In order to conclude it is easy to prove, by using suitable inductive hypotheses that

$$\widehat{ID_1} \vdash \mathcal{I}(\lambda y.c)[\mathcal{I}(a)/x]\,\varepsilon\,\mathcal{I}((\Pi y \in B)\, C[a/x]) \wedge \mathcal{I}(\lambda y.c[a/x])\,\varepsilon\,\mathcal{I}((\Pi y \in B)\, C[a/x]).$$

3. If the judgement $\mathsf{Ap}(\lambda y.b, a) = b[a/x] \in B[a/x]$ is derived in $\mathbf{mTT}^s$ by conversion after having derived $b \in B\,[x \in A]$, $a \in A$, $b[a/x] \in B[a/x]$ and $\mathsf{Ap}(\lambda y.b, a) \in B[a/x]$, in order to show that it is validated by $\mathcal{R}$, we can suppose by inductive hypothesis on validity that $\mathcal{R} \vDash b \in B\,[x \in A]$ and $\mathcal{R} \vDash a \in A$.
By inductive hypothesis on substitution applied to $b \in B\,[x \in A]$ with respect to $a \in A$ we obtain that $\widehat{ID_1} \vdash \mathcal{I}(b)\,[\mathcal{I}(a)/x] \sim_{\mathcal{I}(B\,[a/x])} \mathcal{I}(b\,[a/x])$. But by the definition of the interpretation of $\mathsf{Ap}(\lambda y.b, a)$ from this we obtain that

$$\widehat{ID_1} \vdash \mathcal{I}(\mathsf{Ap}(\lambda y.b, a)) \sim_{\mathcal{I}(B\,[a/x])} \mathcal{I}(b\,[a/x])$$

which exactly means that $\mathcal{R} \vDash \mathsf{Ap}(\lambda y.b, a) = b\,[a/x] \in B\,[a/x]$.

For rules about prop_s, the definitions of $\mathcal{I}(\mathsf{prop}_s)$ and $\mathcal{I}(\tau(p))$ were given in such a way that validity and substitution can be checked easily, sometimes (e. g. in the case of quantifiers $\widehat{(\exists x \in A)}\, p$ and $\widehat{(\forall x \in A)}\, p$) using the inductive hypothesis (coding), which guarantees that if you start from A which is proven to be a set in $\mathbf{mTT}^s$ and you perform the coding in the syntax $\widehat{A}$, then $\widehat{A}$ is a well defined code for a set and it is exactly the internal version of $\mathcal{I}(\mathsf{A})$. ◀

Consequences of the validity theorem

We discuss here about the validity in our realizability model for **mTT** of some principles, namely Extensionality Equality of Functions, Axiom of Choice and formal Church Thesis.

1. **Extensionality Equality of Functions** can be formulated as a proposition in **mTT** as follows:

$$(\mathbf{extFun}) \quad (\forall f \in (\Pi x \in A)\, B)\,(\forall g \in (\Pi x \in A)\, B)\,((\forall x \in A)\,\mathsf{Id}(B, \mathsf{Ap}(f, x), \mathsf{Ap}(g, x)) \to \mathsf{Id}((\Pi x \in A)\, B, f, g))$$

Since the judgements $f = g \in (\Pi x \in A)\, B$ and $\mathsf{Ap}(f, x) = \mathsf{Ap}(g, x) \in B\,[x \in A]$ have the same interpretation, **extFun** can be realized by the term $\Lambda f.\Lambda g.\Lambda r.0$, i. e. our model realises **extFun**.

2. The **Axiom of Choice** $\mathbf{AC}_{A,B}$ is represented in **mTT** by the following proposition:

$$(\mathbf{AC}_{A,B})\ (\forall x \in A)\,(\exists y \in B)\,\rho(x,y) \to (\exists f \in (\Pi x \in A)\,B)\,(\forall x \in A)\,\rho(x, \mathsf{Ap}(f,x))$$

Unfortunately if $\rho(x,y)$ is a proposition for which $\mathcal{R} \vDash \rho(x,y)\,prop\,[x \in A,\, y \in B]$, a realizer r for $(\forall x \in A)\,(\exists y \in B)\,\rho(x,y)$ cannot be turned into a recursive function from $\mathcal{J}(A)$ to $\mathcal{J}(B)$ respecting equivalence relations $\sim_{\mathcal{I}(A)}$ and $\sim_{\mathcal{I}(B)}$, as the interpretation of propositions is proof-irrelevant and we can have different elements a and a' of $\mathcal{J}(A)$ which are equivalent in $\mathcal{I}(A)$ for which $\pi_1(\{r\}(a))$ and $\pi_1(\{r\}(a'))$ are not equivalent in $\mathcal{I}(B)$. This problem can be avoided if A is a numerical set and in particular in the case of the set N. In this case the natural number $\Lambda r.\langle \Lambda n.\pi_1(\{r\}(n)), \Lambda n.\pi_2(\{r\}(n))\rangle$ is a realizer for the axiom of choice $\mathbf{AC}_{N,B}$. So $\mathcal{R} \Vdash \mathbf{AC}_{N,B}$ for every B.
Moreover also the axiom of unique choice $\mathbf{AC}_!$ given by

$$(\mathbf{AC}_!)\ (\forall x \in A)\,(\exists! y \in B)\,\rho(x,y) \to (\exists f \in (\Pi x \in A)\,B)\,(\forall x \in A)\,\rho(x, \mathsf{Ap}(f,x))$$

is validated by the model $\mathcal{R}$.[9] In fact if ρ is a proposition for which we have that $\mathcal{R} \vDash \rho(x,y)\,prop\,[x \in A,\, y \in B]$, then in particular

$$\widehat{ID_1} \vdash \forall x \forall x' \forall y \forall t\,(x \sim_{\mathcal{I}(A)} x' \wedge y\,\varepsilon\,\mathcal{J}(B) \wedge t \Vdash \rho(x,y) \to t \Vdash \rho(x',y)).$$

This implies that we can easily choose a realizer for the axiom of unique choice.

3. If φ is a formula of first-order arithmetic HA, then we can define a proposition $\overline{\varphi}$ in **mTT**, according to the following conditions:

$\overline{\bot}$ is $\bot$	$\overline{\varphi \wedge \varphi'}$ is $\overline{\varphi} \wedge \overline{\varphi'}$	$\overline{\varphi \to \varphi'}$ is $\overline{\varphi} \to \overline{\varphi'}$	$\overline{\exists x\,\varphi}$ is $(\exists x \in N)\,\overline{\varphi}$
$\overline{t = s}$ is $\mathsf{Id}(N, \overline{t}, \overline{s})$	$\overline{\varphi \vee \varphi'}$ is $\overline{\varphi} \vee \overline{\varphi'}$		$\overline{\forall x\,\varphi}$ is $(\forall x \in N)\,\overline{\varphi}$

where $\overline{t}$ and $\overline{s}$ are the translations of terms of HA in **mTT** (in particular primitive recursive functions of HA are translated via El_N, *succ* and 0 are translated in the obvious corresponding ones and variables are interpreted as themselves[10]). The language of HA can also be naturally interpreted in $\widehat{ID_1}$ by using the fact that each primitive recursive function can be encoded by a numeral. If t is a term of HA we will still write t for its translation in $\widehat{ID_1}$. The following lemma is an immediate consequence of the definition of our realizability interpretation where $\Vdash_k$ denotes Kleene realizability in HA (see [20]):

▶ **Lemma 8.** *If t is a term of* HA *and φ is a formula of* HA*, then*
(a) $\widehat{ID_1} \vdash \mathcal{I}(\overline{t}) = t$,
(b) $\widehat{ID_1} \vdash n \Vdash_k \varphi \leftrightarrow n \Vdash \overline{\varphi}$.

The **formal Church Thesis CT** can be expressed in **mTT** as the following proposition

$$(\mathbf{CT})\,(\forall x \in N)\,(\exists y \in N)\,\rho(x,y) \to (\exists e \in N)\,(\forall x \in N)\,(\exists u \in N)\,(\overline{T(e,x,u)} \wedge \rho(x, \overline{U(u)})$$

where T and U are the Kleene predicate and the primitive recursive function representing Kleene application in HA. Note that the validity of **CT** can be obtained by glueing

[9] $(\exists! x \in A)P(x)$ is defined as $(\exists x \in A)P(x) \wedge (\forall x \in A)(\forall x' \in A)(P(x) \wedge P(x') \to \mathsf{Id}(A,x,x'))$.
[10] Here we suppose that variables of HA coincides with variables of the untyped syntax of $\mathbf{mTT}^s$.

$\mathbf{AC}_{N,N}$ together with the following restricted form of Church Thesis for type-theoretic functions:

$$(\mathbf{CT}_\lambda)\,(\forall f \in (\Pi x \in N)N)\,(\exists e \in N)\,(\forall x \in N)\,(\exists u \in N)\,(\overline{T(e,x,u)} \wedge \mathsf{Id}(N, \mathsf{Ap}(f,x), \overline{U(u)}))\,.$$

We know by general results on Kleene realizability that there exists a numeral $\mathbf{r}$ for which $\mathsf{HA} \vdash \exists u\, T(f,x,u) \to (\{\mathbf{r}\}(f,x) \Vdash \exists u\, T(f,x,u))$. Using this remark, the fact that $\{f\}(x)\downarrow$ is equivalent to $\exists u\, T(f,x,u)$ in $\widehat{ID_1}$, the proof irrelevance and lemma 8 we can show that $\mathbf{CT}_\lambda$ can be realized by

$$\Lambda f.\langle f, \Lambda x.\langle \{\mathbf{p}_1\}(\{\mathbf{r}\}(f,x), \langle \{\mathbf{p}_2\}(\{\mathbf{r}\}(f,x), 0\rangle\rangle\rangle.$$

In fact every function from N to N is interpreted in the model as a code for a total recursive function and we can send this code to itself in order to realize Church Thesis. *Proof irrelevance allows to ignore the problem that different codes can give rise to extensionally equal functions, which is crucial to prove validity of* **CT**.
We can conclude this section by stating the following consistency results:

▶ **Theorem 9.** ***mTT*** *is consistent with* ***CT***.

▶ **Corollary 10.** ***emTT*** *is consistent with* ***CT***.

Proof. According to the interpretation of **emTT** in **mTT** in [9], the interpretation of **CT** turns now to be equivalent to **CT** itself. Therefore a model showing consistency of **mTT** with **CT** can be extended to a model of **emTT** with **CT**. ◀

4 Conclusions

As explained in the introduction, the semantics built here seems to be the best approximation of Kleene realizability for the extensional level **emTT** of the Minimalist Foundation, since **emTT** validates Extensionality Equality of Functions and it is constructively incompatible with the Axiom of Choice on generic sets (see [9]), which is instead valid in Beeson's model. In our semantics instances of the axiom of choice are still valid only on numerical sets, which include the interpretation of basic intensional types as the set of natural numbers.

On the contrary, for the intensional level **mTT** of the Minimalist Foundation we hope to build a more intensional realizability semantics à la Kleene where we validate not only **CT** but also the Axiom of Choice **AC** on generic types. Recalling from [9] that our **mTT** can be naturally interpreted in Martin-Löf's type theory with one universe, such an intensional Kleene realizability for **mTT** could be obtained by modelling intensional Martin-Löf's type theory with one universe (with explicit substitutions in place of the usual substitution term equality rules) together with **CT**. However, as far as we know, the consistency of intensional Martin-Löf's type theory with **CT** is still an open problem.

Acknowledgements. We acknowledge many useful fruitful discussions with Takako Nemoto on realizability models for Martin-Löf's type theory during her visits to our department. We also thank Laura Crosilla, Giovanni Sambin and Thomas Streicher for other interesting fruitful discussions on topics of this paper. We are grateful to Ferruccio Guidi for its constant help with typesetting.

References

1 G. Barthes, V. Capretta, and O. Pons. Setoids in type theory. *J. Funct. Programming*, 13(2):261–293, 2003. Special issue on "Logical frameworks and metalanguages".
2 M. Beeson. *Foundations of Constructive Mathematics*. Springer-Verlag, Berlin, 1985.

3 E. Bishop. *Foundations of Constructive Analysis*. McGraw-Hill Book Co., 1967.
4 T. Coquand. Metamathematical investigation of a calculus of constructions. In P. Odifreddi, editor, *Logic in Computer Science*, pages 91–122. Academic Press, 1990.
5 S. Feferman. Iterated inductive fixed-point theories: application to Hancock's conjecture. In *Patras Logic Symposion*, pages 171–196. North Holland, 1982.
6 M. Hofmann. *Extensional Constructs in Intensional Type Theory*. Distinguished Dissertations. Springer, 1997.
7 J. M. E. Hyland. The effective topos. In *The L.E.J. Brouwer Centenary Symposium (Noordwijkerhout, 1981)*, volume 110 of *Stud. Logic Foundations Math.*, pages 165–216. North-Holland, Amsterdam-New York, 1982.
8 J. M. E. Hyland, P. T. Johnstone, and A. M. Pitts. Tripos theory. *Bull. Austral. Math. Soc.*, 88:205–232, 1980.
9 M. E. Maietti. A minimalist two-level foundation for constructive mathematics. *Annals of Pure and Applied Logic*, 160(3):319–354, 2009.
10 M. E. Maietti and G. Rosolini. Elementary quotient completion. *Theory and Applications of Categories*, 27(17):445–463, 2013.
11 M. E. Maietti and G. Rosolini. Quotient completion for the foundation of constructive mathematics. *Logica Universalis*, 7(3):371–402, 2013.
12 M. E. Maietti and G. Rosolini. Unifying exact completions. *Applied Categorical Structures*, DOI 10.1007/s10485-013-9360-5, 2013.
13 M. E. Maietti and G. Sambin. Toward a minimalist foundation for constructive mathematics. In L. Crosilla and P. Schuster, editor, *From Sets and Types to Topology and Analysis: Practicable Foundations for Constructive Mathematics*, number 48 in Oxford Logic Guides, pages 91–114. Oxford University Press, 2005.
14 M. E. Maietti and G. Sambin. Why topology in the Minimalist Foundation must be pointfree. *Logic and Logical Philosophy*, 22(2):167–199, 2013.
15 P. Martin-Löf. *Notes on Constructive Mathematics*. Almqvist & Wiksell, 1970.
16 P. Martin-Löf. *Intuitionistic Type Theory. Notes by G. Sambin of a series of lectures given in Padua, June 1980*. Bibliopolis, Naples, 1984.
17 B. Nordström, K. Petersson, and J. Smith. *Programming in Martin Löf's Type Theory*. Clarendon Press, Oxford, 1990.
18 P. Odifreddi. *Classical recursion theory*, volume 125 of *Studies in Logic and the Foundations of Mathematics*. North-Holland Publishing Co., 1989.
19 E. Palmgren. Bishop's set theory. Slides for lecture at the TYPES summer school, 2005.
20 A. S. Troelstra and D. van Dalen. Constructivism in mathematics, an introduction, vol. I. In *Studies in logic and the foundations of mathematics*. North-Holland, 1988.

Investigating Streamless Sets

Erik Parmann

University of Bergen
Norway
Erik.Parmann@ii.uib.no

Abstract

In this paper we look at *streamless sets*, recently investigated by Coquand and Spiwack [4]. A set is *streamless* if every stream over that set contain a duplicate. It is an open question in constructive mathematics whether the Cartesian product of two streamless sets is streamless.

We look at some settings in which the Cartesian product of two streamless sets is indeed streamless; in particular, we show that this holds in Martin-Löf intentional type theory when at least one of the sets have decidable equality. We go on to show that the addition of functional extensionality give streamless sets decidable equality, and then investigate these results in a few other constructive systems.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Type theory, Constructive Logic, Finite Sets

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.187

1 Introduction

One of the interesting aspects of working in constructive mathematics is that notions often become more nuanced than they do in classical mathematics. This holds for the notion of finiteness, for instance; there are a multitude of possible definitions of a set being finite which would be equivalent classically, but are different constructively.

In this paper, we will look at a particular definition of finite sets in a constructive context, given in terms of streamless sets. This is essentially a constructive version of the classical statement that a set is finite if there are no injections from $\mathbb{N}$ into it. It is formulated positively: a set A is *streamless* when

$$\forall f : \mathbb{N} \to A, \exists i, j : \mathbb{N}, i < j \wedge f(i) = f(j).$$

It is not known who first looked at finiteness in a constructive setting, but it was recently investigated by Coquand and Spiwack [4], who look at four different definitions of a set being finite. These four are, in decreasing order of strength:

- Enumerated: there is a list containing all the elements in the set;
- Bounded: there is an $n : \mathbb{N}$ such that every list with more than n elements has duplicates;
- Noetherian: no matter how one adds elements from the set to a list, one eventually gets duplication in the list; and
- Streamless: every stream over the set contains duplicates.

They show that these notions form a strict hierarchy, except in the streamless and noetherian cases (where strictness is left open): they show that any noetherian set is streamless, and conjecture that the converse does not hold.

It is relatively easy to see that not all bounded sets are enumerated: with enumerated sets we actually have all of its elements, while with bounded sets we only know a bound on the size of the set. In fact, emptiness is in general undecidable for bounded sets, but decidable

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 187–201

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

for enumerated sets. Coquand and Spiwack [4] cite an example offered by F. Richman of a way to generate subsets of natural numbers with the property that one cannot *a priori* know the size of the subset, but if one gets any element in the set then one knows the size of the set. These sets are noetherian but not bounded.

Marc Bezem and other authors have a model of Martin-Löf Type Theory [6] in which there is a streamless set which is not provably noetherian, thus showing that the noetherian property is strictly stronger than streamlessness (personal communication, September 2014). This model is rather complicated and has yet to be published. The authors construct a set parameterized by a undecidable predicate on $\mathbb{N}$. Equality on this set is decidable, which is important for the proof that it is streamless. They assume Markov's principle, and use that as the "engine" which finds duplicates in the streams. They are also able to show that this set cannot be noetherian. In this way they show – since Markov's principle is consistent with Type Theory – that it is not possible to prove that streamlessness implies noetheriannes.

In addition to giving the hierarchy, Coquand and Spiwack [4] also prove several closure properties of the different notions of finiteness. They show that all four are closed under sum; that is, for any of the notions of finiteness, the sum of two finite sets is itself finite by the same notion. The situation is more complicated for Cartesian products. The two strongest notions, enumerated and bounded, are shown to be closed under products, and noetherian sets are closed under products, as long as one of the sets has decidable equality. The use of decidable equality in one of the sets in the proof in [4] was first pointed out in [2]. Whether streamless sets are closed under Cartesian products was left as an open problem.

Our main result will be the following: in Martin-Löf intentional type theory (ITT) [5] streamlessness is closed under Cartesian products, granted that one of the sets has decidable equality or is bounded.

An important feature of ITT is strong Σ-elimination. Consequently, from a proof of $\forall x \exists y.\phi(x, y)$ we are able to get, for any x, an actual y which can be used in the construction of new functions/streams. This plays an important role in the proof of our main result. In other systems, like HA which we will look at in Section 6, we need to assert a axiom of choice to get the same.

In Coq we have the choice of formalizing statements either in Set, which enjoy strong Σ-elimination, or Prop which does not. The proof we provide here will, on the face of it, only hold when streamlessness is formalized in Set; but we will see that, as long as *both* sets have decidable equality, the two formalizations actually correspond.

Decidable equality plays an important role in our proof, and we conjecture that streamlessness is not closed under products when both sets have undecidable equality. We show that, in ITT with functional extensionality, streamless sets have decidable equality, meaning that a potential counter-model must reject functional extensionality.

The main motivation behind this work is curiosity as to the strength of streamless sets, but there is also potential for practical applications. One such example is outlined in Coquand and Spiwack [4], namely automaton reachability testing. They give the regular depth-first graph algorithm for finding reachable states, and then proceed to show that if one assumes that the set of states in the automaton is finite in the sense of streamless, then this algorithm terminates. It is not uncommon to take the Cartesian product of two automata to create one which has as its language the intersection of the two original languages. Given that streamlessness is closed under product, one can show that the reachability algorithm also terminates on this new automaton.

In Section 2, we introduce streamless sets and some machinery which lets us find any number of duplicate elements. In Section 3, we prove the main theorem: that streamless sets

with decidable equality are closed under Cartesian products in ITT. In Section 4, we see that adding functional extensionality gives streamless sets decidable equality. Section 5 relates our findings to Coq and its Set vs Prop distinction; it also briefly touches upon Homotopy Type Theory with Univalence. In Section 6, we relate our finding to Heyting arithmetic in the systems (E-) HA^{ω}. Section 7 provides a brief overview of related works; Section 8 highlights some remaining questions; and we conclude in Section 9.

1.1 Notation

We work in Martin-Löf intensional type theory (ITT) [5], where both propositions and sets are modeled as types.

We assume a inductive type $\mathbb{N}$ for the natural numbers, and we have the usual type constructors: If A is a type and B is a type family over A then both $\Pi_{x:A}B(x)$ and $\Sigma_{x:A}B(x)$ are types, the dependent function type and the dependent pair type with the usual computation rules. We use $\pi_1 : (\Sigma_{x:A}B(x)) \to A$ and $\pi_2 : \Pi_{p:\Sigma_{x:A}B(x)}B(\pi_1(p))$ as the two projections of dependent pairs. In the special cases where $B(x)$ does not depend on x, we abbreviate $\Pi_{x:A}B(x)$ as $A \to B$ and $\Sigma_{x:A}B(x)$ as $A \times B$, the latter being the Cartesian product of A and B. If A and B are types, then $A + B$ is their disjoint union with the constructors $\mathrm{inl} : A \to A + B$ and $\mathrm{inr} : B \to A + B$.

We will use the notation $\mathrm{Dec}\ =_A$ to stand for the type $\Pi_{x:A}\Pi_{y:A}(I_A(x,y) + \neg I_A(x,y))$, where $I_A(x,y)$ is the inductive identity type. We will use $=_A$ as an infix version of I_A, or just $=$ if the type A is clear from context. With A having decidable equality we mean that we have an inhabitant of $\mathrm{Dec}\ =_A$.

A stream over a set A is any function of type $\mathbb{N} \to A$. Given a stream $g : \mathbb{N} \to A$ we also have "cut" streams $g|_n : \mathbb{N} \to A$ for every $n : \mathbb{N}$ defined by

$$g|_n(x) := g(x + n).$$

When we say that we have duplicates in a stream $g : \mathbb{N} \to A$, we mean that we have two indices $i < j$ such that $g(i) =_A g(j)$.

Given a stream g over $A \times B$, we can project out two streams $g_1 : \mathbb{N} \to A$ and $g_2 : \mathbb{N} \to B$ being $g_i = \pi_i \circ g$. As usual, two elements in $A \times B$ are equal if both their first and second projection are equal. We also say that two elements in $A \times B$ are A-equal (resp. B-equal) if their first (resp. second) projections are equal.

2 Introduction to streamless sets

A set A is *streamless* if all streams over it contains duplicates; that is, for all streams $g : \mathbb{N} \to A$, we have indices $i < j$ with $g(i) =_A g(j)$. Formally, it means that we have a inhabitant of the type

$$\mathrm{Streamless}(A) := \Pi_{f:\mathbb{N}\to A}\Sigma_{p:\mathbb{N}\times\mathbb{N}}(\pi_1(p) < \pi_2(p) \times f(\pi_1(p)) =_A f(\pi_2(p))).$$

In what follows we will mostly be interested in the pair $p : \mathbb{N} \times \mathbb{N}$, and not the proof that it has the desired features. To avoid having to project out the number and clutter up the construction more than needed, we will assume that if we have a streamless set A, we have a witness

$$M_A : (\mathbb{N} \to A) \to \mathbb{N} \times \mathbb{N},$$

which, given a stream g over A, gives out two indices $i < j$ such that $g(i) = g(j)$.

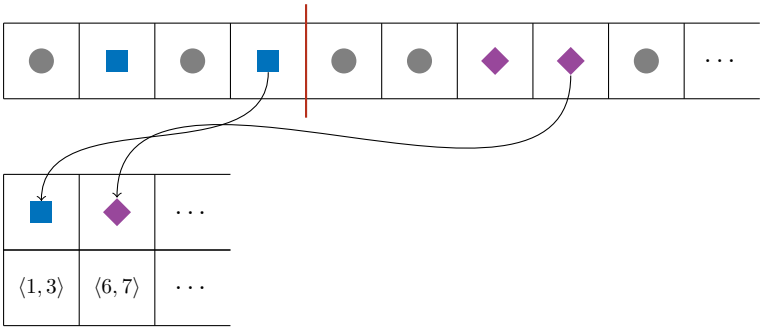


Figure 1 g^2, the stream of duplicates in g.

First, we show that if we have a stream over a streamless set B, we can find not only duplicates, but for any n we can find elements occurring at least n times. This is clear classically; we just have to look at the first $|B| \times n$ elements in the stream. Constructively it is less clear, as we do not know the actual size of the set – only that it is streamless. As seen in the introduction, one cannot, in general, deduce the size of a set from the fact that it is streamless. The first part of this construction, for $n = 2$, is also used to prove that streamless is closed under sum in [4].

Given a stream g over streamless B, we make a new stream g^2 over $B \times \mathbb{N} \times \mathbb{N}$, such that for every $\langle b, i, j\rangle$ we have $i < j$ and $g(i) = g(j) = b$, and for all $g^2(n) = \langle b, i_1, j_1\rangle$ and $g^2(n+1) = \langle c, i_2, j_2\rangle$ we have $j_1 < i_2$. We get this by letting g^2 begin with $\langle g(j), i, j\rangle$ where $\langle i, j\rangle = M_B(g)$, and then continue likewise on the stream $g|_{j+1}$.

Formally, g^2 is defined as follows, where _ indicates a value which we do not use (and thus prefer not to name).

▶ **Definition 1** ($g^2 : \mathbb{N} \to B \times \mathbb{N} \times \mathbb{N}$).

$$
\begin{aligned}
g^2(0) &= \langle g(j), i, j\rangle && \text{where } \langle i, j\rangle = M_B(g),\\
g^2(n+1) &= \langle g(j+p), i+p, j+p\rangle && \text{where } \langle \_, \_, p\rangle = g^2(n)\\
& && \text{and } \langle i, j\rangle = M_B(g|_{p+1})
\end{aligned}
$$

Figure 1 contains a visual representation of g^2, the top being g and the bottom g^2. The two blue boxes make up the first duplicate pair found by $M_B(g)$. The vertical red line indicates that this is where we "cut" the stream, and by using M_B again on this new stream we get a new duplicate pair, the purple diamonds. This process continues, defining a new stream of representatives of duplicates in g.

The first projection of g^2 is itself a B-stream, and we can then use the same process on this stream. This provides duplicate duplicates, giving us elements which occur four times in g.

We can iterate this process and, for every $n : \mathbb{N}$ and stream $g : \mathbb{N} \to B$, we get a stream $g^n : \mathbb{N} \to B \times (List\ \mathbb{N})$ such that every element in the new stream gives a $\langle b, l\rangle$ such that b occurs at least n times in g, at the n different indices given in l.

To formally define g^n it is easiest to first define a slightly stronger function $f^n : \mathbb{N} \to B \times (List\ \mathbb{N}) \times \mathbb{N}$. The last natural number is used when defining $f^{m+1}(n+1)$, it tells it

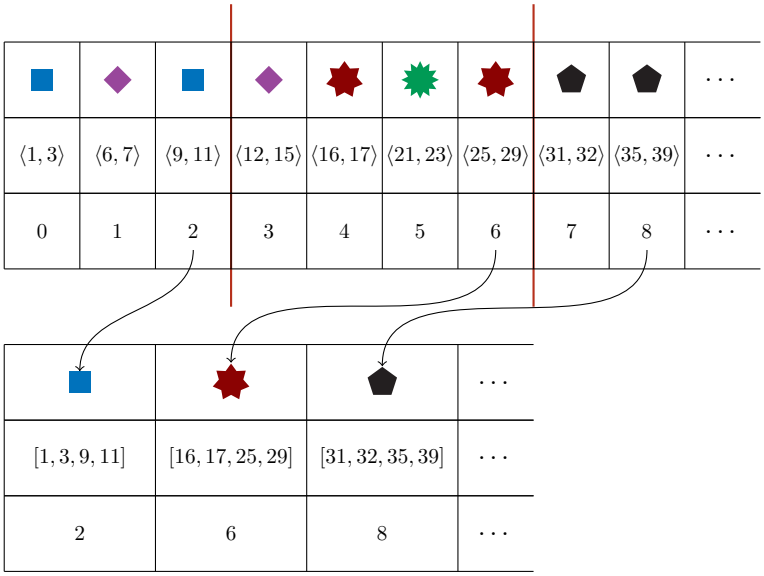


Figure 2 Calculating f^3 from f^2.

where in $(f^m)_1$ the n^{th} duplicate was found, enabling us to cut $(f^m)_1$ at the right place.

$$\begin{aligned} f^2(n) &= \langle b, [i_1, i_2], i_2 \rangle && \text{where } \langle b, i_1, i_2 \rangle = g^2(n) \\ f^{m+1}(0) &= \langle ((f^m)_1(i), (f^m)_2(i) + +(f^m)_2(j), j \rangle && \text{where } \langle i, j \rangle = M_B((f^m)_1), \\ f^{m+1}(n+1) &= \langle ((f^m)_1(i), (f^m)_2(i) + +(f^m)_2(j), j \rangle && \text{where } \langle \_, \_, p \rangle = f^{m+1}(n) \\ & && \text{and } \langle i, j \rangle = M_B((f^m)_1|_p) \end{aligned}$$

This process is illustrated in Figure 2, where we show how to calculate f^3 from f^2. Having f^n we define g^n by simply dropping the third number:

▶ **Definition 2** ($g^n : \mathbb{N} \to B \times List\ \mathbb{N}$).

$$g^n(x) = \langle e, l \rangle \qquad \text{where } \langle e, l, \_ \rangle = f^n(x)$$

The attentive reader notices that this does not actually produce linearly many indices, but exponentially many. g^3 is actually a stream of items occurring 4 times, and g^4 is a stream of objects occurring 8 times. We will not make use of this property, and we will, for the sake of simplicity, assume that g^n contains elements that occur n times.

Observe that we use strong $\exists$ elimination for this construction. Not only do we know that there are indices, but we know what they are; we are also free to use them in the construction of a new stream, to which we can apply M_B once more. As mentioned above, [4] uses a stream which is the first projection of g^2 in the proof that streamless is closed under sum. We do not know of a proof that streamlessness is closed under sum which does not assume strong $\exists$ elimination.

3 Products of streamless sets

This section applies the machinery developed in the previous section to the product of streamless sets.

We will first see that the Cartesian product of a bounded set with a streamless set is streamless. It is worth noting that this is independent of whether any of the sets has decidable equality or not.

▶ **Lemma 3.** *In ITT we have: If at least one of A and B is bounded and the other is streamless then $A \times B$ is streamless.*

Proof. We assume that A is bounded by n. (If it were B then the construction below would be "mirrored".) Given the stream $g : \mathbb{N} \to A \times B$ we look at $g_2 : \mathbb{N} \to B$, its second projection. By looking at $(g_2)^{n+1}(0)$ we get a pair $\langle b, [i_0, \ldots, i_n]\rangle$ such that b occurs at all the indices $i_0, \ldots, i_n$ in g_2. Note that $g_1(i_0), \ldots, g_1(i_n)$ are $n+1$ elements of A, so since A is bounded by n, there must be at least two indices $i_k < i_l$ such that $g_1(i_k) = g_1(i_l)$. As $g_2(i_k) = b = g_2(i_l)$, we get $g(i_k) = g(i_l)$. ◀

We now show that Markov's principle and decidable equality of one of the sets imply that streamlessness is closed under product. This result is a warm up for the later, more general result shown in Theorem 6. The proofs have interesting similarities, especially in how we can use the streamlessness of a set to "emulate" Markov's principle.

First a reminder of Markov's principle.

▶ **Definition 4** (Markov's principle)**.** For decidable predicates P on $\mathbb{N}$ we have $\neg\neg\Sigma_{x:\mathbb{N}}P(x) \to \Sigma_{x:\mathbb{N}}P(x)$.

Markov's principle has a quite computational flavour, which unsurprisingly makes it easier to prove a set streamless. All we need to do to find the duplicate indices is to show that it cannot be the case that they do not exist.

▶ **Lemma 5.** *In ITT with MP we have: If at least one of A and B has decidable equality and A and B are both streamless then $A \times B$ is streamless.*

Proof. We assume a stream $g : \mathbb{N} \to A \times B$. We also assume, without loss of generality, that A is the set with decidable equality. (If it were B then the construction below would be "mirrored".)

We define the following predicate on $\mathbb{N}$:

$$P(n) := \text{For } \langle \_, [i_0, \ldots, i_{n-1}]\rangle = (g_2)^n(0) \text{ we have duplicates in } [g_1(i_0), \ldots, g_1(i_{n-1})].$$

Remember that g_2 gets the B-stream, and $(g_2)^n$ finds n indices with equal elements. Note that if A has decidable equality, $P(n)$ is decidable.

We now proceed to show that (1) $\neg\neg\exists n P(n)$ and that (2) from $\exists n P(n)$ we can get $\langle i, j\rangle$, with $i < j$ and $g(i) = g(j)$.

Proof of (1). We assume $\neg\exists n P(n)$ and proceed to produce a contradiction. $\neg\exists n P(n)$ implies $\forall n \neg P(n)$, which says that for any n we have that for $\langle b, [i_0, \ldots, i_{n-1}]\rangle = (g_2)^n(0)$ the list $[g_1(i_0), \ldots, g_1(i_{n-1})]$ has no duplicates. Notice that the list $[g_1(i_0), \ldots, g_1(i_{n-1})]$ has n elements, all from A.

We now make a duplicate-free stream $f : \mathbb{N} \to A$; that is, for every $n : \mathbb{N}$, we have for all $j < n$ that $f(n) \neq f(j)$, contradicting that A is streamless. Defining $f(n)$ we first find $n+1$ indices with the same b element, $\langle \_, [i_0, \ldots, i_n]\rangle = (g_2)^{n+1}(0)$. We now let

$$f(n) = ([g_1(i_0), \ldots, g_1(i_n)] \setminus [f(0), \ldots f(n-1)])(0).$$

That is, $f(n)$ is the first element in the list resulting from removing any element from $[g_1(i_0), \ldots, g_1(i_n)]$ that the stream already contains. As $[g_1(i_0), \ldots, g_1(i_n)])$ contains $n+1$

different elements from A, we know that the resulting list is non-empty. Since this stream only outputs elements which have not been output up until that point, it will never introduce a duplicate pair. Thus we have contradicted that A is streamless, enabling us to conclude $\neg\neg\exists n P(n)$. ◀

Proof of (2). From $\exists n P(n)$ we have that there is an n such that for $\langle b, [i_0, \ldots, i_{n-1}]\rangle = (g_2)^n(0)$ the list $[g_1(i_0), \ldots, g_1(i_{n-1})]$ has duplicates. Let those indices be $i_k < i_l$. Since every element in $g(i_0), \ldots, g(i_{n-1})$ has b as its second coordinate, we get that $g(i_k) = g(i_l)$. ◀

Having proved $\neg\neg\exists n P(n)$, we apply Markov's principle and get $\exists n P(n)$. By (2) above, this gives us the indices $\langle i, j\rangle$ with $i < j$ and $g(i) = g(j)$. ◀

We will now proceed to get rid of Markov's principle. Several parts of the previous proof will be recognisable, but we use the streamlessness of the two underlying sets to do the work that Markov's principle did in the previous proof.

▶ **Theorem 6.** *In ITT we have: If at least one of A and B has decidable equality and A and B are both streamless, then $A \times B$ is streamless.*

Proof. We assume that A is the set with decidable equality, and we want to construct

$$M_{A\times B} : (\mathbb{N} \to A \times B) \to \mathbb{N} \times \mathbb{N},$$

which, given any $A \times B$-stream g, finds a pair of indices $i < j$ such that $g(i) = g(j)$.

Given an $A \times B$-stream g, we inductively define an A-stream f by letting $f(n)$ first look at $f(m)$ for all $m < n$, and see if two equal elements are outputted. This can be done since A has decidable equality. If there is a duplicate element, $f(n)$ outputs it. If there are no duplicates outputted so far, we let $f(n)$ look at all the A-elements corresponding to the n B-elements given by $(g_2)^n(0)$. Remember, $(g_2)^n(0) = \langle b, l\rangle$ where $l = [i_0, \ldots, i_{n-1}]$ is a list of n indices. Looking up these indices in g_1 gives us a list $la : List\ A$ of n A-elements.

By using the decidability of A, we can check whether there are duplicate elements in la. In the case of no duplicates, we know that there must be at least one of the n elements which does not already occur in f so far (as we have only produced $n-1$ elements so far). We can check which one this is, as we have already defined f up to $n-1$. We then let $f(n)$ be one of those elements which has not occurred in f so far. More precisely,

$$f(n) = ([g_1(i_0), \ldots, g_1(i_{n-1})] \setminus [f(0), \ldots f(n-1)])(0).$$

If, on the other hand, there is some duplicate element in the list, we let $f(m)$ be that element for all $m \geq n$. Notice that if this is the case, this is the first time a duplicate is introduced in f. This completes the construction of $f : \mathbb{N} \to A$, and we will now use f to find duplicates in $g : \mathbb{N} \to A \times B$.

From the construction of f, we have the following property:

▶ **Lemma 7.** *For the smallest i such that $f(i) = f(i+1)$ we have duplicates in the list $[g(l_0) \ldots, g(l_{i-1})]$, where $\langle b, [l_0 \ldots, l_{i-1}]\rangle = (g_2)^i(0)$.*

As A is streamless and f is a A-stream, we can use M_A to find indices $k < d$ of duplicates in f. Since A had decidable equality, we can do a bounded search downward from k to find the first index i such that $f(i) = f(i+1)$. By Lemma 7, we have duplicates in $[g(l_0) \ldots, g(l_{i-1})]$ where $\langle b, [l_0 \ldots, l_{i-1}]\rangle = (g_2)^i(0)$. Thus, we have two indices $l_k < l_m$ in l such that $g_1(l_k) = g_1(l_m)$. By construction all the indices in $[l_0 \ldots, l_{i-1}]$ are B-equal, so $g_2(l_k) = g_2(l_m)$, giving $g(l_k) = g(l_m)$. ◀

Finally observe that if it were B and not A that had decidable equality, the construction above would be "mirrored"; f would have to be a B-stream, and we would use $(g_1)^n(0)$ instead of $(g_2)^n(0)$.

As an example of the construction, let us look at a particular calculation of $f(4)$, where no duplicates have been found so far. That means that so far f looks like

$$f = a_0, a_1, a_2$$

with none of them being equal any other.

$(g_2)^4(0)$ is $\langle b, [n_0, n_1, n_2, n_3\rangle]$, giving four indices in g with the same b-element. This means that g looks somewhat like

$$g = \ldots, \langle b, a'_0\rangle \ldots, \langle b, a'_1\rangle, \ldots, \langle b, a'_2\rangle, \ldots, \langle b, a'_3\rangle, \ldots$$

By the decidability of A, we can check whether there is a duplicate among $[a'_0, a'_1, a'_2, a'_3]$. If not, then we know that there is some element in $[a'_0, a'_1, a'_2, a'_3] \setminus [a_0, a_1, a_2]$, and we let $f(4)$ be the first such element. If there are duplicates, e.g $a'_1 = a'_3$, we let $f(n) = a'_1$ for all $n \geq 4$.

Comparing this proof with the proof using Markov's principle we see that we can use the streamlessness of one of the underlying sets to search for the n which gives us A-duplicates. The trick is to control exactly when duplicates are introduced in the f-stream, and then use the streamlessness of A to recover this point.

We combine Lemma 3 and Theorem 6 to get the following corollary.

▶ **Corollary 8.** *In ITT we have: If at least one of A and B has decidable equality or is bounded, and A and B are both streamless, then $A \times B$ is streamless.*

4 Streamlessness and decidable equality

It should be clear by now that decidable equality of the underlying set is quite important for the ability to produce streamless sets; we will see another indication of this in this section. We will show that in ITT, functional extensionality give streamless sets decidable equality. In addition to showing the close relation between finiteness and decidable equality, it is relevant to the search for a potential counter-model to the claim that streamlessness is closed under Cartesian products even without decidable equality.

As a warm-up, we look at the situation where the set is not only streamless, but bounded. Remember that this means that we have an $n : \mathbb{N}$ such that, for every A-list of more than n elements, we can find a duplicate pair. Formally, this means that we have an inhabitant of the type

$$\text{Bounded}(A) := \Sigma_{n:\mathbb{N}}(\Pi_{l:listA}(\text{len}(l) > n \to \Sigma_{i,j:\mathbb{N}}(i < j \times l[i] = l[j]))).$$

If we want to determine whether a_1 is equal to a_2 we make a list l of $n + 1$ instances of a_1, and get a pair of indices $i_1 < j_1$ with duplicates in this list. We then proceed to swap the element at $l[i_1]$ with a_2, giving a new list. The original list is equal the new list if and only if $a_1 = a_2$.

We then proceed to get two indices $i_2 < j_2$ of duplicate elements in this new list. If this process is assumed to be a function, and thus provide equal outputs for equal inputs, we get $\langle i_1, j_1\rangle = \langle i_2, j_2\rangle$ if and only if $a_1 = a_2$; and since equality on $\mathbb{N}$ is decidable, we are done.

Our proof turned on the facts that (1) the second projection of a witness of Bounded(A) is a function, (2) this function can be assumed to respect equality on its input, and (3) two lists are equal if and only if they are pointwise equal.

We will now mirror this with streamless sets. One major difference between lists and streams is the following: while lists are equal whenever their elements are equal, this only holds for streams if we assume so. It is consistent to assume an inhabitant of the following type in ITT, and if we do so for all types, we say that we have *functional extensionality*.

▶ **Definition 9** (FunExt(A))**.**

$$\mathrm{FunExt}(A) := \Pi_{f,g:\mathbb{N}\to A}(\Pi_{n:\mathbb{N}}(f(n) =_A g(n)) \to f =_{\mathbb{N}\to A} g)\,.$$

▶ **Lemma 10.** *In ITT with functional extensionality we have: If A is streamless then it has decidable equality.*

Proof. We assume an inhabitant of FunExt(A) and two elements $a, b : A$, and we proceed to determine their equality. Let the stream f_a be the constant A-stream consisting of only a, and let $\langle i, j\rangle$ be the indices returned by $M_A(f_a)$. We now make the stream f'_a which is constantly a, except at index i, where it is b:

$$f'_a(n) = \begin{cases} b & \text{if } n =_{\mathbb{N}} i \\ a & \text{otherwise} \end{cases}$$

Notice that if $a =_A b$ we have $\Pi_{n:\mathbb{N}} f_a(n) =_A f'_a(n)$, so from functional extensionality we then have $f_a = f'_a$. So, by functionality of M_A, we get $a = b \to M_A(f_a) = M_A(f'_a)$, and thus

$$M_A(f_a) \neq M_A(f'_a) \to a \neq b.$$

Concluding, if $M_A(f'_a) \neq \langle i, j\rangle$ then $a \neq b$, and if $M_A(f'_a) = \langle i, j\rangle$ then $a = b$ (as $f'_a(i) = b$ and $f'_a(j) = a$), and since equality on $\mathbb{N}$ is decidable we are done. ◀

Lemma 10 is relevant for the search of a counter-model to the general claim that streamlessness is closed under product. From section 3, we know that such a counter-model must have two streamless sets with undecidable equality. This section shows that the model must also reject functionality extensionality for us to have a streamless set with undecidable equality.

It also highlights some of the difficulty of defining finiteness for sets with undecidable equality in a computational setting, and since the other notions of finiteness given in [4] imply streamlessness, this result also covers them. All the definitions of finiteness have some sort of equality/duplication check at their core. Given this it seems plausible that a proof of finiteness can, in certain situations, lead to decidability. On the other hand, it is quite unsatisfactory that, in certain settings, we are unable to define finite sets of elements with undecidable equality.

In the next section we look at how to formalize both this and the previous results in Coq.

5 Formalization in Coq and HoTT

5.1 Coq: Prop and Set

In this section we will relate the above results to the proof assistant Coq [3], where we have to deal with the distinction between Prop and Set. Functions, which is how we defined streams, live in the universe Set, while there is a separate universe Prop for propositions. The intention is, roughly, to separate between types where we care about the internal structure of the inhabitants (Set) and where we care only about the existence of the inhabitant (Prop).

Given a inhabitant of a type in Prop one is generally not allowed to eliminate on it to construct elements in Set; thus we can not build the new stream g_2 of duplicates using indexes found from a witness of a type in Prop. This means that the constructions given in this paper can not be implemented in Coq *as they stand* if streamless is written as follows:

```
Definition StreamlessEx(A:Set):= forall g:nat → A,
          exists i j, i<j ∧ g(i) = g(j).
```

One way to remedy the situation is to define the notion of a set being streamless in the following way, closer to the way it was encoded in ITT. The notation "$\{x : nat \mid P(x)\}$" is Coq's notation for $\Sigma_{x:\mathbb{N}}P(x)$.

```
Definition StreamlessSig (A:Set):= forall g:nat→ A,
          {ij : nat*nat | fst ij < snd ij ∧ g(fst ij)=g(snd ij)}.
```

StreamlessSig enables us to use the proof of a set being streamless in a computation; in particular it enables us to construct the stream g_2 needed to prove Corollary 8 in Coq. The disadvantage is that it can make it harder to prove sets to be streamless in the first place. There is reason to believe that there are fewer sets satisfying StreamlessSig than StreamlessEx.

In general, whether one wants the statement in Prop or in Set reflects whether one wants to work proof relevant or not; formalizing it as StreamlessSig enables us to use the proof (of a set being streamless) in a computation.

StreamlessSig A implies StreamlessEx A, while the provability of the converse implication is unknown. Interestingly, it *is* know for sets with decidable equality, since we are able to prove the following lemma in Coq for A with decidable equiality, making the two notions of streamless coincide in those cases.

```
Lemma streamlessExToStrSig(A:Set)(A_dec: DecidableEq A) :
                      StreamlessEx A → StreamlessSig A
```

Essential for the proof is the following lemma, holding for decidable predicates P on $\mathbb{N}$, and shown in the Coq library *Coq.Logic.ConstructiveEpsilon*[1].

```
Lemma constructive_indefinite_ground_description_nat :
      (exists x : nat, P x) → {x : nat | P x}.
```

With the indefinite ground description the proof is straightforward. We assume that we have some pairing/decoding functions enabling us to encode pairs of natural numbers as single natural numbers. We then define versions of both StreamlessEx and StreamlessSig using single numbers, prove that the single and paired versions are equivalent, and then it is a simple application of the indefinite ground description given above.

The conclusion is the following corollary:

▶ **Corollary 11.** *In Coq we can prove that StreamlessEx (and StreamlessSig) of sets with decidable equality is closed under Cartesian products.*

A natural question is whether we can strengthen this to say that StreamlessEx is closed under Cartesian products as long as *at least one* of the sets have decidable equality. Unfortunately, this does not follow from the current construction. To see this, assume an $A \times B$-stream g. The construction in Proof 3 uses $(g_2)^n$ to find n-indices with B-equal

[1] http://coq.inria.fr/library/Coq.Logic.ConstructiveEpsilon.html

elements. But for this to be definable in Coq using the technique above, B needs decidable equality. The proof then uses the decidability of A to eliminate on whether there are duplicates among the resulting A-elements or not. It does not seem possible to manipulate the construction such that it is enough for only one of the sets to have decidable equality.

We are also able to reproduce Lemma 10 in Coq for StreamlessSig, the proof is simply a direct Coq formalization of the proof given in Section 4.

```
Lemma strSigAndFuncExtImpliesDecA (A:Set) (Ma:StreamlessSig A)
    (fext: functional_extensionality nat A): forall a b :A, {a=b}+{not(a=b)}.
```

Again, we are not able to simply adapt the proof to StreamlessEx, since the proof crucially uses the indexes returned from M_A in the construction of new functions.

All the Lemmas in this section have been formalized and proved in Coq[2].

5.2 HoTT

Closely related to the Prop/Set distinction is the truncated and non-truncated statements one encounters in Homotopy Type Theory (HoTT). Truncation is a type former which "truncates" a type – removing all information contained in the inhabitants of that type except their existence – and it is written as $||A||$ for a type A. (For more information we refer the reader to the freely available book [10].) We will not go further into HoTT here; but what is relevant for us is that we have a HoTT version of the indefinite ground description above. For decidable predicates P we have

$$||\Sigma_{n:\mathbb{N}} P(n)|| \to \Sigma_{n:\mathbb{N}} P(n)$$

as stated by exercise 3.19 in [10]. One should be able to reproduce a version of Corollary 11 in this setting, getting that for the non-truncated version of streamless it is enough for one of the sets to have decidable equality for streamlessness to be closed under Cartesian products.

With our current knowledge we need both sets to have decidable equality for the truncated version to be closed under Cartesian products without further assumptions, and we conjecture that this is in fact a strict requirement. If we choose to assume the HoTT-version of the axiom of choice,

$$(\Pi_{x:X} ||\Sigma_{a:A(x)} P(x,a)||) \to ||\Sigma_{g:\Pi_{(x:X)} A(x)} \Pi_{x:X} P(x, g(x))||,$$

we can show that truncated-streamless sets are closed under products as long as one of the sets has decidable equality.

In HoTT we can also assume the Univalence axiom, giving that isomorphic structures can be identified. Importantly, the univalence axiom implies functional extensionality. Lemma 10 makes it clear that – unless we want every streamless set to have decidable equality – we must use the truncated version of streamlessness in this setting.

6 HA$^\omega$

It is natural to ask how closely coupled the above results are to the particular constructive setting we are working in, and whether we can reproduce them in a different setting. We will now look at how the results fit in the system HA$^\omega$, an extension of Heyting Arithmetic to the language of finite types, see [9] for more information on HA$^\omega$.

[2] The Coq-script can be found at `https://github.com/epa095/streamless-in-coq`.

HA^ω is proof-irrelevant and does not have strong Σ elimination; instead, we have to use the axiom of choice to extract a function, giving the witnesses which we can then use as terms in the logic.

The set of finite types $\mathcal{T}$ is built from the basic type 0 ($\mathbb{N}$) and is closed under $\times$ and $\to$. HA^ω is "neutral" in the terminology of [9]; we do not assume decidability of $=_\tau$ for any other types than 0, nor do we assume that equality between functions is extensional.

Sets are not a primitive notion in HA^ω, so when talking about sets we mean functions of the type $A : \tau \to 0$; such functions represent the set of elements on which it returns 1. This means that all sets will have decidable membership and sets can only contain elements of one and the same type. For a set $A : \tau \to 0$, we call τ the enclosing type of A. Following [9] we will write $a < b$ in place of $<(a, b) = 1$, where the latter is the characteristic function of the less-than relation. With "a stream over A" we mean a function $f^{0\to\tau}$ where τ is the enclosing type of A such that $\forall n^0(A(f(n)) = 1)$.

Streamlessness of A_τ in this setting is expressed as

$$\mathrm{Streamless}(A_\tau) := \forall g^{0\to\tau}((\forall n^0 A(g(n)) = 1) \to \exists i^0 j^0(i < j \wedge g(i) = g(j))).$$

In order to formalize our results in HA^ω, we first need to define some axioms. $\mathrm{AC}_{\sigma,\tau}$ is the following axiom schema,

$$\mathrm{AC}_{\sigma,\tau} := \forall x^\sigma \exists y^\tau \phi(x, y) \to \exists z^{\sigma\to\tau} \forall x^\sigma \phi(x, zx),$$

and AC is the axiom schema consisting of $\mathrm{AC}_{\sigma,\tau}$ for all types $\sigma, \tau \in \mathcal{T}$. $\mathrm{EXT}_{\sigma,\tau}$ is the following axiom schema,

$$\mathrm{EXT}_{\sigma,\tau} := \forall y^{\sigma\to\tau} z^{\sigma\to\tau}((\forall x^\sigma, yx = zx) \to y = z),$$

and if we add $\mathrm{EXT}_{\sigma,\tau}$ for all types $\sigma, \tau \in \mathcal{T}$ we get the system E-HA^ω.

To reproduce the proof of Lemma 5 in HA^ω, we need to construct the function g^2 of duplicates, and for this we need access to, for every stream, a pair of indexes with duplicate elements in that stream. The following instance of AC for every type τ enclosing a streamless set is enough to mirror Lemma 5 in HA^ω.

$$\mathrm{AC}_{0\to\tau,0} := \forall x^{0\to\tau} \exists y^0 \phi(x, y) \to \exists z^{(0\to\tau)\to 0)} \forall x^\sigma \phi(x, zx)$$

Let the $\phi(x, y)$ stand for the predicate "$(\forall i^0 A(x(i)) = 1) \to y$ encodes a pair of indexes $i < j$ such that $x(i) = x(j)$". Then the antecedent of $\mathrm{AC}_{0\to\tau,0}$ follows immediately from A being streamless, and the result is the function M_A, needed to reconstruct the machinery in the proof of Lemma 5.

▶ **Corollary 12.** *In* $\mathrm{HA}^\omega + \mathrm{AC}$ *we have that streamless sets are closed under products.*

Encoding sets by their characteristic functions yields decidable membership, but in general not decidable equality. The extensionality of E-HA^ω, giving that streams are equal when they are pointwise equal, enables us to mirror Lemma 10:

▶ **Corollary 13.** *In* E- $\mathrm{HA}^\omega + \mathrm{AC}$ *we have that streamless sets have decidable equality.*

Note that E- $\mathrm{HA}^\omega + \mathrm{AC}$ does not prove the law of excluded middle, as it is conservative over HA. For further details, see [1].

7 Related work

One of the first investigations of streamlessness known to the author is by Richman and Stolzenberg [8]. In their terms, a streamless set is called **2**-good, where **2** is the set of two-element subsets of the natural numbers. They show that the sum of two **B**-good sets, of which **2**-good is an instance, is **B**-good, but leave it open for products. This paper does not resolve any of their open questions, as they work in a more general setting than equality. They also give another notion, that of a set being *bar-good*, and they show that the Cartesian product of a bar-good set with a **B**-good set is **B**-good. It is not clear what the relation between streamlessness and bar-good is, and whether there are natural axioms one can assume to make a streamless set bar-good.

Veldman and Bezem [11] investigate the constructive content of the Ramsey theorem [7], giving a constructive proof of a reformulation of it. For this, they use what they call *almost-full* binary relations; relations R on $\mathbb{N}$ where, for every *increasing* function $f : \mathbb{N} \to \mathbb{N}$,

$$\exists m, n : \mathbb{N}, m < n \wedge R(f(m), f(n)).$$

They postulate the axiom of bar-induction, and with it they prove that almost-full relations are closed under intersection. They name this the Intuitionistic Ramsey Theorem, and show that it is classically equivalent to Ramsey's Theorem.

Using equality as the relation R, one gets a notion which comes quite close to streamlessness, apart from Veldman and Bezem's requirement that the functions are increasing, and the fact that streamlessness is a concept applicable for any type (not only $\mathbb{N}$), possibly with undecidable equality.

In light of this, it is natural to ask whether the proofs in this paper can be generalized to relations other than equality. We define what it means for a reflexive and transitive relation R on A to be a well-quasi-ordering:

$$\mathrm{Wqo}_A(R) := \forall g : \mathbb{N} \to A, \exists i, j : \mathbb{N}, i < j \wedge R(g(i), g(j)).$$

Note that a set A is streamless exactly when we have $\mathrm{Wqo}_A(=_A)$. We can ask if the intersection of two such relations is itself a Wqo and whether the proof of Lemma 6 suggest how this could be shown. Unfortunately, we do not see how. We are still able to use the construction g^n to find n elements $a_1, \ldots a_n$ such that $a_1 R a_2 \ldots R a_n$, but we do not have the property that with n elements $b_1, \ldots b_n$ such that *none* of them are R-related, and $n-1$ elements $b'_1, \ldots b'_{n-1}$ such that none of them are R-related, there must be one of the $b_1, \ldots b_n$ which is not R related to any of the $b'_1, \ldots b'_{n-1}$. We have this property when the relation R is equality, and this is used in the proof of Lemma 6.

If we *did* have that Wqo relations were closed under intersection we would immediately get that streamless sets are closed under products: define the relation R_1 on $A \times B$ as $(a_1, b_1) R_1 (a_2, b_2)$ if and only if $a_1 =_A a_2$, and likewise for R_2, looking at the second projections. If A and B are streamless sets, then R_1 and R_2 are Wqo relations and their intersection is equality on $A \times B$.

Vytiniotis, Coquand and Wahlstedt [12] provide an inductive formulation of almost full relations on arbitrary types. They show – if we instantiate their proofs with the relation being equality – that it implies streamlessness, and show that almost-full relations are closed under intersection.

Streamlessness works in a quite general setting, with few assumptions on the underlying set. Bezem et al. [2] impose further restrictions, and the result is a interesting hierarchy of finiteness notions. The restrictions imposed are that equality is decidable; that the subset is

defined by some decidable predicate; and that the set is a subset of some set that can be enumerated. This holds for decidable subsets of natural numbers in particular. The authors find six different formalizations and put them into a hierarchy.

8 Remaining questions

There are several questions remaining. The main one is whether one can show that streamlessness is closed under Cartesian products in ITT without assuming decidable equality. Secondly, to what degree can one show similar results in systems without strong Σ elimination – for example, for StreamlessEx in Coq or the truncated statement in HoTT? And what is the relationship between StreamlessEx and StreamlessSig for sets with undecidable equality?

We conjecture that there exists a model showing that, in ITT, the product of two streamless sets with undecidable equality is not necessarily streamless. From Lemma 10 we know that such a model must reject functional extensionality, and from Lemma 3, we know that neither of the sets can be bounded.

At this point there are, to this author's knowledge, only two sets which are known to be streamless but not bounded. One is the set presented in [4], originally suggested by F. Richman, showing that not all noetherian sets are bounded. As noetherian sets are streamless, this is also a streamless set. But this set has the interesting property that, once one looks at any of the elements in the set, one knows the size of the set! So it is not bounded *a priori*, but if one is given a stream of elements from the set, one can deduce its size and then continue as in the proof of Lemma 3.

The second set, presented in the still unpublished article by Bezem et al. showing that not all streamless sets are noetherian, does not have this property. On the other hand, it has decidable equality, rendering it useless as a counter-model. There does not seem to be an easy way to tweak the model to get rid of this decidable equality; it is essential for the proof that the set is streamless as the authors use Markov's Principle to find the duplicate pair, and Markov's Principle is only applicable for decidable predicates.

To conclude, we currently have no good candidate for a streamless set with a non-streamless Cartesian product. Constructing a suitable streamless set, non-bounded and with undecidable equality, appears to be quite complicated. Neither of the ways used to prove a set streamless – that is, by gathering information about the size of the set encoded in the elements themselves, or using Markov's principle – is likely to work. It seems the most promising route to a counter-model involves finding novel ways to construct streamless sets.

Lastly, we would like to encourage other to look for new notions of finiteness, especially trying to find notions that works nicely and robustly for sets with undecidable equality.

9 Conclusion

We showed that, in Martin-Löf intensional type theory, if at least one of the streamless sets A and B has decidable equality or is bounded, then the Cartesian product $A \times B$ is streamless. We also saw that adding functional extensionality to ITT gives streamless sets decidable equality; and we mirrored these results in both (E-) $\mathrm{HA}^{\omega} + \mathrm{AC}$ and in Coq.

Acknowledgements. I want to thank Arnaud Spiwack and Thierry Coquand for their valuable feedback and questions after my presentation on this topic at TYPES 2014, and Marc Bezem for both introducing this problem to me and discussing it with me. I also wish

to thank the anonymous reviewers for their challenging feedback which led to significant changes to this paper, and Maja Jaakson which kindly proofread it.

References

1. Michael Beeson. Goodman's theorem and beyond. *Pacific Journal of Mathematics*, 84(1):1–16, 1979.
2. M. Bezem, K. Nakata, and T. Uustalu. On streams that are finitely red. *Logical Methods in Computer Science*, 2011. `http://www.cs.ioc.ee/~keiko/papers/finiteness.pdf`.
3. The Coq Development Team. *The Coq Reference Manual, version 8.4*, August 2012. Available electronically at `http://coq.inria.fr/doc`.
4. Thierry Coquand and Arnaud Spiwack. Constructively finite? In *Contribuciones científicas en honor de Mirian Andrés Gómez*, pages 217–230. Universidad de La Rioja, 2010.
5. Per Martin-Löf. An intuitionistic theory of types: predicative part. In H. E. Rose and J. C. Shepherdson, editors, *Proceedings of the Logic Colloquium 1973*, volume 80 of *Studies in Logic and the Foundations of Mathematics*, pages 73–118. North-Holland, 1975.
6. Per Martin-Löf and Giovanni Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, 1984.
7. F. P. Ramsey. On a problem of formal logic. *Proceedings of the London Mathematical Society*, 2(1):264–286, 1930.
8. F. Richman and G. Stolzenberg. Well quasi-ordered sets. *Advances in Mathematics*, 97(2):145–153, 1993.
9. Anne Sjerp Troelstra and Dirk Van Dalen. *Constructivism in mathematics*, volume 2. Elsevier, 1988.
10. The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.
11. Wim Veldman and Marc Bezem. Ramsey's theorem and the pigeonhole principle in intuitionistic mathematics. *Journal of the London Mathematical Society*, 2(2):193–211, 1993.
12. Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. Stop when you are almost-full. In *Interactive Theorem Proving*, pages 250–265. Springer, 2012.

Nominal Presentation of Cubical Sets Models of Type Theory

Andrew M. Pitts

University of Cambridge Computer Laboratory, Cambridge CB3 0FD, UK
andrew.pitts@cl.cam.ac.uk

Abstract

The cubical sets model of Homotopy Type Theory introduced by Bezem, Coquand and Huber [2] uses a particular category of presheaves. We show that this presheaf category is equivalent to a category of sets equipped with an action of a monoid of name substitutions for which a finite support property holds. That category is in turn isomorphic to a category of nominal sets [15] equipped with operations for substituting constants 0 and 1 for names. This formulation of cubical sets brings out the potentially useful connection that exists between the homotopical notion of *path* and the nominal sets notion of *name abstraction*. The formulation in terms of actions of monoids of name substitutions also encompasses a variant category of cubical sets with diagonals, equivalent to presheaves on Grothendieck's "smallest test category" [8, pp. 47–48]. We show that this category has the pleasant property that path objects given by name abstraction are exponentials with respect to an interval object.

1998 ACM Subject Classification F.4.1 Mathematical Logic, D.3.3 Language Constructs and Features

Keywords and phrases models of dependent type theory, homotopy type theory, cubical sets, nominal sets, monoids

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.202

1 Introduction

We begin by reviewing the notion of *cubical set* introduced by Bezem, Coquand and Huber [2, Section 2] and motivate the move to a nominal presentation of it.

▶ **Definition 1.1** (Directions)**.** Throughout the paper $\mathbb{D}$ is a fixed, infinite set whose elements we write as $\mathsf{x}, \mathsf{y}, \mathsf{z}, \ldots$ and call *directions*. We assume $\mathbb{D}$ is disjoint from the two-element set $2 = \{0, 1\}$.

Let $\mathbf{C}$ be the small category whose objects $X, Y, \ldots$ are finite subsets of $\mathbb{D}$ and whose hom-sets $\mathbf{C}(X, Y)$ consist of all functions $s : X \to Y \cup 2$ with the property

$$(\forall \mathsf{x}, \mathsf{x}' \in X)\ s\,\mathsf{x} = s\,\mathsf{x}' \notin 2 \ \Rightarrow\ \mathsf{x} = \mathsf{x}' \tag{1}$$

Such a function is extended to one defined on the whole of $X \cup 2$ by taking $s\,0 = 0$ and $s\,1 = 1$. Then composition in $\mathbf{C}$ is given by composition of functions, with identity morphisms given by inclusions $X \hookrightarrow X \cup 2$.

▶ **Definition 1.2** (Cubical sets)**.** A *cubical set* is a functor $\mathbf{C} \to \mathbf{Set}$ and a morphism of cubical sets is a natural transformation between such functors. Thus the category of cubical sets is the category $[\mathbf{C}, \mathbf{Set}]$ of set-valued presheaves on the category $\mathbf{C}^{\mathrm{op}}$.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 202–220

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

We refer the reader to Bezem, Coquand and Huber [2] for the geometric intuition behind the terminology 'cubical set' and the connection with the notion of the same name in homotopy theory. Like any presheaf category, $[\mathbf{C}, \mathbf{Set}]$ gives rise to a model of extensional Martin-Löf Type Theory, organised as a *category with families* (CwF) in the sense of Dybjer [5]. See Hofmann [9, Section 4] for an account of presheaf CwFs. In order to model Homotopy Type Theory and in particular Voevodsky's Univalence Axiom [21], Bezem, Coquand and Huber consider families of presheaves equipped with operations for filling open boxes – a more uniform version of the classic Kan filling condition in combinatorial homotopy theory. The resulting families of *Kan cubical sets* support an interpretation of identity types and [2] contains a sketch of why there is a universe satisfying the Univalence Axiom with respect to these identity types.

Motivation for a nominal approach

Presheaf models of type theory in general, and in particular the cubical sets model of Homotopy Type Theory mentioned above, inevitably involve quantifications over Kripke possible-worlds (which are finite sets of directions in the cubical case) that tend to obscure the simple intuition behind these models, because of the need to write explicit weakening functions from a world to future worlds. Furthermore, cubical sets of paths and the Kan filling condition make use of constructions involving a choice of directions $\mathsf{x} \in \mathbb{D}$ that are suitably fresh, but whose properties are independent of which particular fresh direction is chosen. This is precisely the situation for which the theory of *nominal sets* [6, 15] was created. In particular it admits a rich theory of *freshness* that makes implicit the dependence upon possible worlds of directions. According to the authors of the experimental implementation of Kan cubical sets [4], "it was convenient to use the alternative presentation of cubical sets as nominal sets". That alternative presentation was announced in [14]. Here we take an alternative approach based on monoids of name substitutions, leading to the equivalences of Theorems 2.9 and 2.13 below. This facilitates the description of Π-types (Section 3.2) and universes (Section 3.3); but more importantly, it allows path objects to be described in terms of the well-developed nominal sets theory of *name abstraction* (Section 2.2). The presentation in terms of monoids of substitutions also encompasses a variant of cubical sets with diagonals (Section 4), equivalent to presheaves on Grothendieck's "smallest test category" [8, pp. 47–48] and referred to in [2]. We show that this category has the pleasant property that path objects given by name abstraction are exponentials with respect to an interval object (Theorem 4.2).

A note on constructivity

The model of univalence based on simplicial sets [12] uses classical set theory. One of Bezem, Coquand and Huber's motivations for considering cubical sets instead of simplicial sets is that they can be made a model of univalence within constructive logics, which makes a computational version possible. It is therefore of interest whether the results in this paper are constructively valid. Like [15], upon some of whose results it relies, this paper is written using naive classical set theory. In a constructive setting, equality for elements of the set $\mathbb{D}$ of directions should be assumed to be decidable and $\mathbb{D}$ should be 'finitely inexhaustible', in the sense that for each subset $X \subseteq \mathbb{D}$ that is in bijection with a finite ordinal, there exists some $\mathsf{x} \in \mathbb{D}$ with $\mathsf{x} \notin X$. Starting from that basis, it seems likely that much of the theory of nominal sets is constructively valid. However, at the very least one has to replace the use of *smallest* finite support sets in arguments by the existence of *some* finite support set. For if

equality is undecidable in some set upon which name permutations act, then the existence of some finite support for an element of the set does not necessarily mean there is a smallest one; see [20, Section 1.2.1]. We leave for future work the questions of whether use of smallest supports can always be avoided and whether the results of this paper are constructively valid.

2 Monoids of Substitutions

In this section we reformulate cubical sets in terms of monoids of substitutions, where the crucial property of 'finite support' gives a well-behaved theory of degeneracy via freshness.

▶ **Definition 2.1** (Substitutions). As far as this paper is concerned, a *finite substitution* is a function $\sigma : \mathbb{D} \to \mathbb{D} \cup 2$ for which $\mathrm{Dom}\,\sigma \triangleq \{\mathsf{x} \in \mathbb{D} \mid \sigma\,\mathsf{x} \neq \mathsf{x}\}$ is finite. Let $\mathbf{Sb}$ denote the monoid whose elements are finite substitutions, with the monoid operation given by composition: $\sigma\sigma' \triangleq \hat{\sigma} \circ \sigma'$, where $\hat{\sigma} : \mathbb{D} \cup 2 \to \mathbb{D} \cup 2$ is the function

$$\begin{aligned} \hat{\sigma}\,\mathsf{b} &\triangleq \mathsf{b} \quad \text{if } \mathsf{b} \in 2, \\ \hat{\sigma}\,\mathsf{x} &\triangleq \sigma\,\mathsf{x} \quad \text{if } \mathsf{x} \in \mathbb{D}. \end{aligned} \tag{2}$$

(Note that $\mathrm{Dom}\,\sigma\sigma'$ is indeed finite, since it is contained in $\mathrm{Dom}\,\sigma \cup \mathrm{Dom}\,\sigma'$.) The identity element $\iota \in \mathbf{Sb}$ is given by the inclusion $\mathbb{D} \hookrightarrow \mathbb{D} \cup 2$. If $\mathsf{x} \in \mathbb{D}$ and $i \in \mathbb{D} \cup 2$, we write

$$(i/\mathsf{x}) \in \mathbf{M} \tag{3}$$

for the finite substitution mapping x to i and otherwise acting like the identity; and if $\mathsf{x}, \mathsf{x}' \in \mathbb{D}$, then we write

$$(\mathsf{x}\;\mathsf{x}') \in \mathbf{M} \tag{4}$$

for the finite substitution that transposes x and x' and otherwise acts like the identity. By a *monoid of substitutions* $\mathbf{M}$ we mean any submonoid of $\mathbf{Sb}$ containing $(\mathsf{x}\;\mathsf{x}')$ and (b/x) for all $\mathsf{b} \in 2$ and all $\mathsf{x}, \mathsf{x}' \in \mathbb{D}$.

The notion of finite support is most often applied to actions of permutations, for example in the theory of nominal sets [15, Chapter 2]. However, it generalizes well to actions of more general forms of substitution; see [7, Definition 7], for example.

▶ **Definition 2.2** (Finitely supported $\mathbf{M}$-sets). For any monoid $\mathbf{M}$ we write write $\mathbf{Set}^{\mathbf{M}}$ for the category whose objects are sets Γ equipped with a (left) $\mathbf{M}$-action $\_ \cdot \_ : \mathbf{M} \times \Gamma \to \Gamma$

$$\iota \cdot d = d \qquad \sigma' \cdot (\sigma \cdot d) = \sigma'\sigma \cdot d \qquad (d \in \Gamma, \sigma, \sigma' \in \mathbf{M}) \tag{5}$$

and whose morphisms are functions $\gamma : \Gamma \to \Gamma'$ preserving the action

$$\gamma(\sigma \cdot d) = \sigma \cdot (\gamma\, d) \qquad (\sigma \in \mathbf{M}, d \in \Gamma) \tag{6}$$

When $\mathbf{M}$ is a monoid of substitutions (Definition 2.1) and $\Gamma \in \mathbf{Set}^{\mathbf{M}}$, we say that a finite subset $X \subseteq_{\mathrm{fin}} \mathbb{D}$ *supports* an element $d \in \Gamma$ if

$$(\forall \sigma, \sigma' \in \mathbf{M})\; ((\forall \mathsf{x} \in X)\; \sigma\,\mathsf{x} = \sigma'\mathsf{x}) \;\Rightarrow \sigma \cdot d = \sigma' \cdot d \tag{7}$$

We write $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ for the full subcategory of $\mathbf{Set}^{\mathbf{M}}$ consisting of those Γ such that for all $d \in \Gamma$ there exists a finite subset $X \subseteq_{\mathrm{fin}} \mathbb{D}$ that supports d. We call $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ the *category of finitely supported* $\mathbf{M}$*-sets*.

▶ **Example 2.3** (The interval). Let $\mathbf{M}$ be a monoid of substitutions. We make $\mathrm{I} \triangleq \mathbb{D} \cup 2$ into an object of $\mathbf{Set}^{\mathbf{M}}$ via the action given by function application: $\sigma \cdot i \triangleq \hat{\sigma}\, i$, for all $i \in \mathrm{I}$. With respect to this action, an element of $\mathsf{x} \in \mathbb{D} \subseteq \mathrm{I}$ is supported by $\{\mathsf{x}\}$ and the two elements of $2 \subseteq \mathrm{I}$ are supported by $\emptyset$. We call I the *interval* in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$.

▶ **Lemma 2.4.** *Let $\mathbf{M}$ be a monoid of substitutions and $\mathsf{b} \in 2$ some fixed Boolean value. For each $d \in \Gamma \in \mathbf{Set}^{\mathbf{M}}$, a finite subset $X \subseteq_{\mathrm{fin}} \mathbb{D}$ supports d iff*

$$(\forall \mathsf{x} \in \mathbb{D})\ \mathsf{x} \notin X \ \Rightarrow\ (\mathsf{b}/\mathsf{x}) \cdot d = d \tag{8}$$

Proof. Taking $\sigma = (\mathsf{b}/\mathsf{x})$ and $\sigma' = \iota$ in (7), we get that it implies (8). To prove (8) implies (7), we proceed by induction on the size of the finite set

$$\mathrm{ds}(\sigma, \sigma') \ \triangleq\ \{\mathsf{x} \in \mathbb{D} \mid \sigma\,\mathsf{x} \neq \sigma'\mathsf{x}\} \tag{9}$$

(It is finite, because it is contained in $\mathrm{Dom}\,\sigma \cup \mathrm{Dom}\,\sigma'$.) The base case is trivial. For the induction step, suppose

$$(\forall \mathsf{x} \in X)\ \sigma\,\mathsf{x} = \sigma'\mathsf{x} \tag{10}$$

and that $\mathsf{y} \in \mathrm{ds}(\sigma, \sigma')$. We have to prove that $\sigma \cdot d = \sigma' \cdot d$. Since $\sigma\,\mathsf{y} \neq \sigma'\mathsf{y}$, from (10) we must have $\mathsf{y} \notin X$ and hence $(\forall \mathsf{x} \in X)\ \sigma(\mathsf{b}/\mathsf{y})\,\mathsf{x} = \sigma'(\mathsf{b}/\mathsf{y})\,\mathsf{x}$. Since $\mathrm{ds}(\sigma(\mathsf{b}/\mathsf{y}), \sigma'(\mathsf{b}/\mathsf{y})) = \mathrm{ds}(\sigma, \sigma') - \{\mathsf{y}\}$, by induction hypothesis $\sigma(\mathsf{b}/\mathsf{y}) \cdot d = \sigma'(\mathsf{b}/\mathsf{y}) \cdot d$. But since X satisfies (8) and $\mathsf{y} \notin X$, it follows that $(\mathsf{b}/\mathsf{y}) \cdot d = d$. Therefore $\sigma \cdot d = \sigma \cdot ((\mathsf{b}/\mathsf{y}) \cdot d) = \sigma(\mathsf{b}/\mathsf{y}) \cdot d = \sigma'(\mathsf{b}/\mathsf{y}) \cdot d = \sigma' \cdot ((\mathsf{b}/\mathsf{y}) \cdot d) = \sigma' \cdot d$, as required. ◀

▶ **Corollary 2.5.** *Suppose $\Gamma \in \mathbf{Set}^{\mathbf{M}}$ and $d \in \Gamma$ is supported by $X \subseteq_{\mathrm{fin}} \mathbb{D}$.*

1. *For any morphism $\gamma : \Gamma \to \Gamma'$ in $\mathbf{Set}^{\mathbf{M}}$, $\gamma\, d \in \Gamma'$ is also supported by X.*
2. *For any $\sigma \in \mathbf{M}$, $\sigma \cdot d \in \Gamma$ is supported by the finite subset $\sigma X \cap \mathbb{D} = \{\sigma\,\mathsf{x} \mid \mathsf{x} \in X \wedge \sigma\,\mathsf{x} \notin 2\}$.*

Proof. Fix some $\mathsf{b} \in 2$. For part 1, if $\mathsf{x} \in \mathbb{D}$ satisfies $\mathsf{x} \notin X$, then

$$\begin{aligned} (\mathsf{b}/\mathsf{x}) \cdot (\gamma\, d) &= \gamma((\mathsf{b}/\mathsf{x}) \cdot d) && \text{by (6)} \\ &= \gamma\, d && \text{by Lemma 2.4.} \end{aligned}$$

So by Lemma 2.4 again, X supports $\gamma\, d$.

For part 2, if $\mathsf{y} \notin \sigma X \cap \mathbb{D}$, then $(\forall \mathsf{x} \in X)\ (\mathsf{b}/\mathsf{y})\sigma\,\mathsf{x} = \widehat{(\mathsf{b}/\mathsf{y})}(\sigma\,\mathsf{x}) = \sigma\,\mathsf{x}$; so because X supports d we have $(\mathsf{b}/\mathsf{y}) \cdot (\sigma \cdot d) = (\mathsf{b}/\mathsf{y})\sigma \cdot d = \sigma \cdot d$. So Lemma 2.4 implies that $\sigma X \cap \mathbb{D}$ supports $\sigma \cdot d$. ◀

▶ **Definition 2.6** (Least supports). Let $\mathbf{M}$ be a monoid of substitutions and $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$. By Lemma 2.4, for each $d \in \Gamma$

$$\mathrm{supp}\, d \triangleq \{\mathsf{x} \in \mathbb{D} \mid (0/\mathsf{x}) \cdot d \neq d\} \tag{11}$$

is finite and is the least finite supporting set of directions for d. Note that $\mathrm{supp}\, d = \{\mathsf{x} \in \mathbb{D} \mid (1/\mathsf{x}) \cdot d \neq d\}$.

▶ **Definition 2.7** (The monoid $\mathbf{Cb}$). Let $\mathbf{Cb} \subseteq \mathbf{Sb}$ be the subset consisting of finite substitutions σ satisfying an injectivity condition like (1):

$$(\forall \mathsf{x}, \mathsf{x}' \in \mathbb{D})\ \sigma\,\mathsf{x} = \sigma\,\mathsf{x}' \notin 2 \ \Rightarrow\ \mathsf{x} = \mathsf{x}'$$

Cb is a monoid of substitutions in the sense of Definition 2.1. It enjoys the following homogeneity property with respect to the small category **C** from Section 1.

▶ **Lemma 2.8** (Homogeneity). *For all morphism $s \in \mathbf{C}(X,Y)$ there is a finite substitution $\sigma \in \mathbf{Cb}$ satisfying $(\forall \mathsf{x} \in X)\ s\,\mathsf{x} = \sigma\,\mathsf{x}$.*

Proof. Given $s \in \mathbf{C}(X,Y)$, let $X_1 \triangleq \{\mathsf{x} \in X \mid s\,\mathsf{x} \notin 2\}$ and $X_2 \triangleq \{\mathsf{x} \in X \mid s\,\mathsf{x} \in 2\}$. Thus $X = X_1 \uplus X_2$ and s restricts to a bijection between X_1 and $Y_1 \triangleq \{s\,\mathsf{x} \mid \mathsf{x} \in X_1\}$. Pick a finite permutation π of $\mathbb{D}$ that agrees with s on X_1 and is the identity outside the finite set $X_1 \cup Y_1$ (it is always possible to do so – see for example [15, Lemma 1.14]). Then

$$\sigma\,\mathsf{x} \triangleq \begin{cases} s\,\mathsf{x} & \text{if } \mathsf{x} \in X_2 \\ \pi\,\mathsf{x} & \text{otherwise} \end{cases}$$

is a suitable element of **Cb**. ◀

▶ **Theorem 2.9.** *The category $[\mathbf{C}, \mathbf{Set}]$ of cubical sets is equivalent to the category $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ of finitely supported **Cb**-sets.*

Proof. We define a functor $I^* : [\mathbf{C}, \mathbf{Set}] \to \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ as follows. Each inclusion $X \subseteq Y$ between finite subsets of $\mathbb{D}$ yields a morphism $X \hookrightarrow Y$ in **C**. So given $C \in [\mathbf{C}, \mathbf{Set}]$ we can take the colimit of C restricted to the poset $(\mathrm{P}_{\mathrm{fin}}\,\mathbb{D}, \subseteq)$ of finite subsets of $\mathbb{D}$: $I^*C \triangleq \mathrm{colim}_{X \in \mathrm{P}_{\mathrm{fin}}\,\mathbb{D}}\, C\,X$. Concretely, I^*C consists of equivalence classes $[X,x]$ of pairs $(X,x) \in \sum_{X \in \mathbf{C}} C\,X$ for the equivalence relation that relates (X,x) and (X',x') when there is some $Y \supseteq X \cup X'$ with $C(X \hookrightarrow Y)x = C(X' \hookrightarrow Y)x'$. Note that by definition of the monoid **Cb**, for each $\sigma \in \mathbf{Cb}$ and $X \in \mathbf{C}$ the restricted function $\sigma|_X : X \to \sigma X$ is a morphism in $\mathbf{C}(X, \sigma X \cap \mathbb{D})$. Then

$$\sigma \cdot [X,x] \triangleq [\sigma X \cap \mathbb{D}, C(\sigma|_X)x] \tag{12}$$

gives a well-defined **Cb**-action on I^*C. Furthermore, with respect to this action an element $[X,x] \in I^*C$ is supported by X; for if σ and σ' agree on X, then $C(\sigma|_X) = C(\sigma'|_X)$ and hence $\sigma \cdot [X,x] = \sigma' \cdot [X,x]$. So $I^*C \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$.

The assignment $C \in [\mathbf{C}, \mathbf{Set}] \mapsto I^*C \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ extends to a functor as follows. Given a natural transformation $\varphi : C \to C'$ in $[\mathbf{C}, \mathbf{Set}]$ we get a well-defined function $I^*\varphi : I^*C \to I^*C'$ by defining

$$I^*\varphi\,[X,x] \triangleq [X, \varphi_X x] \tag{13}$$

The naturality of φ_X in $X \in \mathbf{C}$ ensures not only that this definition is independent of the choice of representative (X,x) for the element $[X,x]$, but also that $I^*\varphi$ preserves the **Cb**-action (12).

We complete the proof of the theorem by showing that $I^* : [\mathbf{C}, \mathbf{Set}] \to \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ is faithful, full and essentially surjective.

- *I^* is faithful*: Note that any inclusion $X \hookrightarrow Y$ in **C** is split, for example by the morphism $p \in \mathbf{C}(Y,X)$ where

  $$p\,\mathsf{y} \triangleq \begin{cases} \mathsf{y} & \text{if } \mathsf{y} \in X \\ 0 & \text{otherwise} \end{cases} \qquad (\mathsf{y} \in Y)$$

  Therefore $C(X \hookrightarrow Y) : C\,X \to C\,Y$ is an injective function in **Set** with left inverse $C\,p$. Thus if $\varphi, \varphi' \in [\mathbf{C}, \mathbf{Set}](C, C')$ and $I^*\varphi = I^*\varphi'$, then for any $X \in \mathbf{C}$ and $x \in C\,X$ we have $[X, \varphi_X x] = I^*\varphi\,[X,x] = I^*\varphi'[X,x] = [X, \varphi'_X x]$, so that for some $Y \supseteq X$, $C(X \hookrightarrow Y)(\varphi_X x) = C(X \hookrightarrow Y)(\varphi'_X x)$; and since $C(X \hookrightarrow Y)$ is injective this gives $\varphi_X x = \varphi'_X x$. Therefore $\varphi = \varphi'$.

- I^* *is full*: Suppose $C, C' \in [\mathbf{C}, \mathbf{Set}]$ and $\gamma \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}(I^*C, I^*C')$. It is not hard to see that

$$(\forall d \in I^*C')\ \mathrm{supp}\, d \subseteq X \ \Rightarrow\ (\exists! x \in C'\, X)\ d = [X, x] \tag{14}$$

Indeed if $d = [Y, y]$ and $\mathrm{supp}\, d \subseteq X$, then d is supported by $X \cap Y$ and hence the substitution $\sigma \in \mathbf{Cb}$ mapping each $\mathsf{y} \in Y - X$ to 0 and otherwise acting like the identity satisfies $\sigma \cdot d = d$; therefore $d = \sigma \cdot [Y, y] = [X \cap Y, C'(\sigma|_Y)y] = [X, C'(X \cap Y \hookrightarrow X)C'(\sigma|_Y)y]$. The uniqueness part of (14) follows from the injectivity of each $C'(X \hookrightarrow Y)$, noted above. For each $X \in \mathbf{C}$ and $x \in C\, X$, by part 1 of Corollary 2.5 we have that $\mathrm{supp}(\gamma[X, x]) \subseteq \mathrm{supp}[X, x]$ and hence that $\mathrm{supp}(\gamma[X, x]) \subseteq X$. Therefore from (14) we have $(\forall x \in C\, X)(\exists! x' \in C'X)\ \gamma[X, x] = [X, x']$. So for each $X \in \mathbf{C}$ there is a function $\varphi_X : C\, X \to C'X$ satisfying

$$(\forall x \in C\, X)\ \gamma[X, x] = [X, \varphi_X x] \tag{15}$$

It suffices to show that φ_X is natural in X, since then by combining (13) with (15) we have that $\varphi \in [\mathbf{C}, \mathbf{Set}](C, C')$ satisfies $I^*\varphi = \gamma$. For naturality, given $s \in \mathbf{C}(X, Y)$ to prove $\varphi_Y(C\, s\, x) = C's\, (\varphi_X x)$ it suffices to show $[Y, \varphi_Y(C\, s\, x)] = [Y, C's\, (\varphi_X x)]$, because of the injectivity of the functions $C(Y \hookrightarrow Z) : C\, Y \to C\, Z$ (see above). Now we use the homogeneity property in Lemma 2.8: picking a substitution $\sigma \in \mathbf{Cb}$ that agrees with s on X, we have $[Y, \varphi_Y(C\, s\, x)] = \gamma[Y, C\, s\, x] = \gamma[\sigma X \cap \mathbb{D}, C(\sigma|_X)x] = \gamma(\sigma \cdot [X, x]) = \sigma \cdot (\gamma[X, x]) = \sigma \cdot [X, \varphi_X x] = [\sigma X \cap \mathbb{D}, C'(\sigma|_X)(\varphi_X x)] = [Y, C's\, (\varphi_X x)]$, as required.

- I^* *is essentially surjective*: Given $\Gamma \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$, for each $X \in \mathbf{C}$ consider the subset of Γ consisting of the elements supported by the finite subset $X \subseteq_{\mathrm{fin}} \mathbb{D}$:

$$I_*\Gamma\, X \triangleq \{d \in \Gamma \mid \mathrm{supp}\, d \subseteq X\} \tag{16}$$

For each $s \in \mathbf{C}(X, Y)$ there is a well-defined function $I_*\Gamma\, s : I_*\Gamma\, X \to I_*\Gamma\, Y$ satisfying

$$I_*\Gamma\, s\, d = \sigma \cdot d \quad \text{where } \sigma \in \mathbf{Cb} \text{ is any substitution satisfying } \sigma|_X = s \tag{17}$$

(There is such a σ by Lemma 2.8; $I_*\Gamma\, s\, d$ is independent of the choice of σ because X supports d; and $I_*\Gamma\, s\, d \in I_*\Gamma\, Y$ by part 2 of Corollary 2.5.) Since $\iota|_X = \mathrm{id}_X$ we get $I_*\Gamma\, \mathrm{id}_x\, d = \iota \cdot d = d$; and since $(\sigma'\sigma)|_X = s' \circ s$ when $s = \sigma|_X$ and $s' = \sigma'|_Y$, we get $I_*\Gamma\, (s' \circ s)\, d = \sigma'\sigma \cdot d = \sigma' \cdot (\sigma \cdot d) = I_*\Gamma\, s'(I_*\Gamma\, s\, d)$. So $I_*\Gamma \in [\mathbf{C}, \mathbf{Set}]$. To complete the proof we will construct an isomorphism $\varepsilon_\Gamma : I^*(I_*\Gamma) \cong \Gamma$ in $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$.
First note that in (17), if s is an inclusion $X \hookrightarrow Y$, then we can take $\sigma = \iota$ and therefore $I_*\Gamma\, (X \hookrightarrow Y)d = \iota \cdot d = d$. It follows that if (X, d) and (X', d') both represent the same equivalence class in $I^*(I_*\Gamma)$, then $d = I_*\Gamma\, (X \hookrightarrow Y)d = I_*\Gamma\, (X' \hookrightarrow Y)d' = d'$ (where $Y \supseteq X \cup X'$). So we get a well-defined function $\varepsilon_\Gamma : I^*(I_*\Gamma) \to \Gamma$ satisfying

$$\varepsilon_\Gamma[X, d] = d \tag{18}$$

This preserves the $\mathbf{Cb}$-action because

$$\begin{aligned}
\varepsilon_\Gamma(\sigma \cdot [X, d]) &= \varepsilon_\Gamma[\sigma X \cap \mathbb{D}, I_*\Gamma(\sigma|_X)d] && \text{by (12)} \\
&= I_*\Gamma(\sigma|_X)d && \text{by (18)} \\
&= \sigma \cdot d && \text{by (17)} \\
&= \sigma \cdot (\varepsilon_\Gamma[X, d]) && \text{by (18) again.}
\end{aligned}$$

It is an injective function, because if $[X, d], [X', d'] \in I^*(I_*\Gamma)$ satisfy $d = d'$, then $\mathrm{supp}\, d = \mathrm{supp}\, d' \subseteq X \cap X'$ (by Lemma 2.4) and as above we have $I_*\Gamma(X \cap X' \hookrightarrow X)d = d = d' = I_*\Gamma(X \cap X' \hookrightarrow X')d'$; hence $[X, d] = [X', d']$. It is a surjective function, because each $d \in \Gamma$ is finitely supported by some $X \subseteq_{\mathrm{fin}} \mathbb{D}$ and hence $d = \varepsilon_\Gamma[X, d]$. So altogether, ε_Γ is an isomorphism in $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$. ◀

2.1 Nominal sets with 01-substitution

In this section we deduce from Theorem 2.9 the equivalence announced in [14]. We assume some familiarity with the theory of nominal sets; see for example [15].

Let **Nom** denote the category of nominal sets and equivariant functions over the set of atoms $\mathbb{D}$. If $\mathbf{M}$ is any monoid of substitutions (Definition 2.1), then the group $\mathrm{Perm}\,\mathbb{D}$ of finite permutations of $\mathbb{D}$ is a submonoid of $\mathbf{M}$, because every finite permutation is the composition of finitely many transpositions. Thus the $\mathbf{M}$-action on each $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ restricts to a $\mathrm{Perm}\,\mathbb{D}$-action. If $X \subseteq \mathbb{D}$ supports $d \in \Gamma$ in the sense of Definition 2.2, then it is in particular a support in the usual sense of nominal sets [15, Section 2.1]:

$$(\forall \pi \in \mathrm{Perm}\,\mathbb{D})\ ((\forall \mathsf{x} \in X)\ \pi\,\mathsf{x} = \mathsf{x})\ \Rightarrow\ \pi \cdot d = d \tag{19}$$

Hence each $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ is a nominal set and indeed there is a forgetful functor $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}} \to \mathbf{Nom}$, since morphisms in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ are in particular equivariant functions.

▶ **Lemma 2.10** (Freshness)**.** *Suppose* $\mathbf{M}$ *is a monoid of substitutions and that* $d \in \Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$*. Then* $\mathrm{supp}\,d$ *as defined in Definition 2.6 is the least finite support for* $d \in \Gamma$ qua *nominal sets, that is, the least finite subset* $X \subseteq \mathbb{D}$ *satisfying* (19)*. Hence the relation*

$$\mathsf{x} \# d \ \triangleq\ \mathsf{x} \notin \mathrm{supp}\,d \qquad (\mathsf{x} \in \mathbb{D}, d \in \Gamma)$$

coincides with the nominal notion of freshness [15, Chapter 3].

Proof. First note that since (7) implies (19), $\mathrm{supp}\,d$ is a finite subset of $\mathbb{D}$ satisfying (19). If $X \subseteq_{\mathrm{fin}} \mathbb{D}$ is any other such, we will show that (8) holds and hence that $\mathrm{supp}\,d \subseteq X$, by Lemma 2.4. Indeed, if $\mathsf{b} \in 2$ and $\mathsf{x} \in \mathbb{D} - X$, choose some $\mathsf{y} \in \mathbb{D}$ not in the finite subset $X \cup \{\mathsf{x}\} \cup \mathrm{supp}\,d$. Then

$$\begin{aligned}
(\mathsf{b}/\mathsf{x}) \cdot d &= (\mathsf{x}\ \mathsf{y})(\mathsf{b}/\mathsf{y})(\mathsf{x}\ \mathsf{y}) \cdot d && \text{since } (\mathsf{b}/\mathsf{x}) = (\mathsf{x}\ \mathsf{y})(\mathsf{b}/\mathsf{y})(\mathsf{x}\ \mathsf{y}) \in \mathbf{M} \\
&= (\mathsf{x}\ \mathsf{y})(\mathsf{b}/\mathsf{y}) \cdot d && \text{by (19) with } \pi = (\mathsf{x}\ \mathsf{y}), \text{ since } \mathsf{x}, \mathsf{y} \notin X \\
&= (\mathsf{x}\ \mathsf{y}) \cdot d && \text{by Lemma 2.4, since } \mathsf{y} \notin \mathrm{supp}\,d \\
&= d && \text{by (19) again}
\end{aligned}$$

as required for (8). ◀

▶ Remark 2.11. By contrast with nominal sets in general, the freshness relation for objects of $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ can be characterised in terms of substitution of 0 or 1, as follows:

$$\mathsf{x} \# d \ \Leftrightarrow\ (0/\mathsf{x}) \cdot d = d \ \Leftrightarrow\ (1/\mathsf{x}) \cdot d = d \qquad (\mathsf{x} \in \mathbb{D}, d \in \Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}) \tag{20}$$

This is an immediate consequence of Definition 2.6, which relies upon the characterisation of support in Lemma 2.4.

▶ **Definition 2.12** (Nominal 01-substitution structures)**.** Let **01-Nom** be the category whose objects are nominal sets Γ equipped with *source* and *target* operations $(\mathsf{x} := 0)\_, (\mathsf{x} := 1)\_ : \Gamma \to \Gamma$ in each direction $\mathsf{x} \in \mathbb{D}$ satisfying for all $\pi \in \mathrm{Perm}\,\mathbb{D}$, $\mathsf{x}, \mathsf{x}' \in \mathbb{D}$, $\mathsf{b}, \mathsf{b}' \in 2$ and $d \in \Gamma$

$$\pi \cdot ((\mathsf{x} := \mathsf{b})d) = (\pi\,\mathsf{x} := \mathsf{b})(\pi \cdot d) \tag{21}$$

$$\mathsf{x} \# (\mathsf{x} := \mathsf{b})d \tag{22}$$

$$\mathsf{x} \# d \Rightarrow (\mathsf{x} := \mathsf{b})d = d \tag{23}$$

$$\mathsf{x} \neq \mathsf{x}' \Rightarrow (\mathsf{x} := \mathsf{b})(\mathsf{x}' := \mathsf{b}')d = (\mathsf{x}' := \mathsf{b}')(\mathsf{x} := \mathsf{b})d \tag{24}$$

The morphisms of **01-Nom** are the equivariant functions $\gamma \in \mathbf{Nom}(\Gamma, \Gamma')$ that also commute with the source and target operations in each direction: $\gamma((\mathsf{x} := \mathsf{b})d) = (\mathsf{x} := \mathsf{b})(\gamma\, d)$. Composition and identities are as in **Nom**.

▶ **Theorem 2.13.** *The category* **01-Nom** *of nominal sets with* 01*-substitution structure is isomorphic to the category* $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ *of finitely supported* **Cb***-sets and hence (by Theorem 2.9) is equivalent to the category* $[\mathbf{C}, \mathbf{Set}]$ *of cubical sets.*

Proof. We noted above that the **Cb**-action on $\Gamma \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ restricts to a $\mathrm{Perm}\,\mathbb{D}$-action, making it a nominal set. We get source and target operations in each direction $\mathsf{x} \in \mathbb{D}$ by defining $(\mathsf{x} := \mathsf{b})d \triangleq (\mathsf{b}/\mathsf{x}) \cdot d$. These satisfy (21) because $\pi(\mathsf{b}/\mathsf{x}) = (\mathsf{b}/\pi\,\mathsf{x})\pi \in \mathbf{Cb}$; they satisfy (22) because of part 2 of Corollary 2.5 and Lemma 2.10; they satisfy (23) because of Lemmas 2.4 and 2.10; and they satisfy (24) because $(\mathsf{b}/\mathsf{x})(\mathsf{b}'/\mathsf{x}') = (\mathsf{b}'/\mathsf{x}')(\mathsf{b}/\mathsf{x}) \in \mathbf{Cb}$ when $\mathsf{x} \neq \mathsf{x}'$. Furthermore, since each morphism $\gamma \in \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}(\Gamma, \Gamma')$ commutes with the **Cb**-action, it is not only an equivariant function, but also preserves the source and target operations defined as above. So we get a functor $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}} \to \mathbf{01\text{-}Nom}$ which is the identity on underlying nominal sets.

Conversely, given $\Gamma \in \mathbf{01\text{-}Nom}$, we can combine the $\mathrm{Perm}\,\mathbb{D}$-action with the source and target operations to get a **Cb**-action on Γ as follows: for each $\sigma \in \mathbf{Cb}$ and $d \in \Gamma$, consider

$$\sigma \cdot d \triangleq \pi \cdot (\mathsf{x}_1 := \mathsf{b}_1) \cdots (\mathsf{x}_n := \mathsf{b}_n)d \tag{25}$$

where $\mathsf{x}_1, \ldots, \mathsf{x}_n$ are the distinct element of $\{\mathsf{x} \in \mathrm{Dom}\,\sigma \mid \sigma\,\mathsf{x} \in 2\}$, where $\mathsf{b}_i = \sigma\,\mathsf{x}_i$ for $i = 1, \ldots, n$, and where $\pi \in \mathrm{Perm}\,\mathbb{D}$ is a finite permutation agreeing with σ on $\{\mathsf{x} \in \mathrm{Dom}\,\sigma \mid \sigma\,\mathsf{x} \notin 2\}$. Note that there is such a permutation, because σ is injective on $\{\mathsf{x} \in \mathrm{Dom}\,\sigma \mid \sigma\,\mathsf{x} \notin 2\}$; and (25) is independent of which π we choose, and independent of the order in which we list the elements of $\{\mathsf{x} \in \mathrm{Dom}\,\sigma \mid \sigma\,\mathsf{x} \in 2\}$ (because of property (24)). In case $n = 0$, by $(\mathsf{x}_1 := \mathsf{b}_1) \cdots (\mathsf{x}_n := \mathsf{b}_n)d$ we mean d. Thus $\iota \cdot d = d$; and it is not hard to see that this definition also satisfies $\sigma' \cdot (\sigma \cdot d) = \sigma'\sigma \cdot d$. So we get a **Cb**-action on Γ and clearly each $d \in \Gamma$ is supported by $\mathrm{supp}\, d$ with respect to this action. Furthermore, for each morphism $\gamma \in \mathbf{01\text{-}Nom}(\Gamma, \Gamma')$, since $\gamma(\pi \cdot (\mathsf{x}_1 := \mathsf{b}_1) \cdots (\mathsf{x}_n := \mathsf{b}_n)d) = \pi \cdot (\mathsf{x}_1 := \mathsf{b}_1) \cdots (\mathsf{x}_n := \mathsf{b}_n)(\gamma\, d)$ and $\mathrm{supp}(\gamma\, d) \subseteq \mathrm{supp}\, d$, we get $\gamma(\sigma \cdot d) = \sigma \cdot (\gamma\, d)$. So we get a functor $\mathbf{01\text{-}Nom} \to \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$, which once again is the identity on underlying nominal sets.

It is easy to see that these two functors are mutually inverse, so that $\mathbf{01\text{-}Nom} \cong \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$. ◀

▶ Remark 2.14. The proof of the equivalence $[\mathbf{C}, \mathbf{Set}] \simeq \mathbf{01\text{-}Nom}$ given in [14] is somewhat different from the above one and was inspired by proofs of equivalences between (pre)sheaf categories and nominal sets equipped with substitution structures studied by Staton [17]; see in particular the proof of Proposition 9 in [18].

2.2 Path objects

One of the advantages of replacing cubical sets by the equivalent notion of nominal sets with 01-substitution (Theorem 2.13) is that the construct used in [2, Section 8.2] to model identity types coincides across the equivalence with a central and widely used notion of nominal set theory, namely that of *name abstraction* [15, Chapter 4].

Given a nominal set $\Gamma \in \mathbf{Nom}$, the nominal set $[\mathbb{D}]\Gamma$ of name-abstractions of elements of Γ has underlying set consisting of equivalence classes of pairs $(\mathsf{x}, d) \in \mathbb{D} \times \Gamma$ for a generalised form of α-equivalence, namely the equivalence relation

$$(\mathsf{x}, d) \approx_\alpha (\mathsf{x}', d') \;\triangleq\; (\exists \mathsf{y} \,\#\, (\mathsf{x}, d, \mathsf{x}', d'))\ (\mathsf{y}\ \mathsf{x}) \cdot d = (\mathsf{y}\ \mathsf{x}') \cdot d' \tag{26}$$

We write $\langle \mathsf{x} \rangle d$ for the $\approx_\alpha$-equivalence class of (x, d). The action of finite permutations $\pi \in \operatorname{Perm} \mathbb{D}$ on such equivalence classes is well-defined by

$$\pi \cdot \langle \mathsf{x} \rangle d \triangleq \langle \pi\, \mathsf{x} \rangle (\pi \cdot d) \tag{27}$$

and one can show that the least support of $\langle \mathsf{x} \rangle d$ with respect to this action is $\operatorname{supp} d - \{\mathsf{x}\}$; see [15, Proposition 4.5].

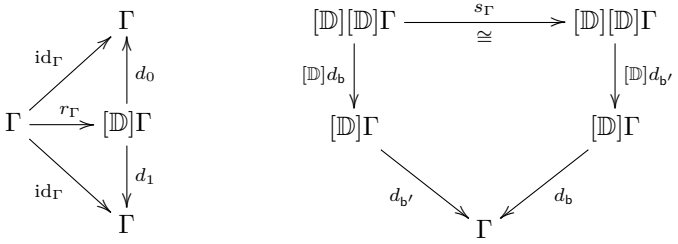
If $\Gamma \in$ **01-Nom**, then because of properties (21) and (22), the source and target operations induce morphisms $d_0, d_1 \in \mathbf{Nom}([\mathbb{D}]\Gamma, \Gamma)$ satisfying

$$d_{\mathsf{b}} \langle \mathsf{x} \rangle d = (\mathsf{x} := \mathsf{b}) d \qquad (\mathsf{b} \in 2, \mathsf{x} \in \mathbb{D}, d \in \Gamma)$$

Then properties (23) and (24) correspond to the commutation of the following diagrams, where $r_\Gamma \in \mathbf{Nom}(\Gamma, [\mathbb{D}]\Gamma)$ and $s_\Gamma \in \mathbf{Nom}([\mathbb{D}][\mathbb{D}]\Gamma, [\mathbb{D}][\mathbb{D}]\Gamma)$ are morphisms satisfying

$$r_\Gamma\, d = \langle \mathsf{x} \rangle d \qquad \text{for some/any } \mathsf{x} \# d \tag{28}$$

$$s_\Gamma \langle \mathsf{x} \rangle \langle \mathsf{y} \rangle d = \langle \mathsf{y} \rangle \langle \mathsf{x} \rangle d \tag{29}$$



In fact these are diagrams in **01-Nom**, because there is a well-defined nominal 01-substitution structure on each $[\mathbb{D}]\Gamma$ satisfying

$$(\mathsf{x} := \mathsf{b})(\langle \mathsf{y} \rangle d) = \langle \mathsf{y} \rangle ((\mathsf{x} := \mathsf{b}) d) \quad \text{if} \quad \mathsf{x} \neq \mathsf{y} \tag{30}$$

and then d_0, d_1, r_Γ and s_Γ are morphisms in **01-Nom**.

For each $\Gamma \in$ **01-Nom**, one can think of elements $p \in [\mathbb{D}]\Gamma$ as *paths* in Γ from $d_0 p$ to $d_1 p$. For each $d \in \Gamma$, $r_\Gamma\, d \in [\mathbb{D}]\Gamma$ is a degenerate path from d to itself. The object $\langle d_0 \,,\, d_1 \rangle : [\mathbb{D}]\Gamma \to \Gamma \times \Gamma$ of the slice category **01-Nom**$/\Gamma \times \Gamma$ corresponds under the equivalence of Theorem 2.13 to the structure that Bezem, Coquand and Huber use to model identity types, at least in the case that the cubical set corresponding to Γ satisfies a uniform Kan filling condition [2, Section 5].

3 Modelling Type Theory with Families of M-sets

We have reformulated cubical sets in a way that emphasises actions of monoids of substitutions. Since any monoid $\mathbf{M}$ can be regarded as a one-object category, $\mathbf{Set}^{\mathbf{M}}$ is in particular a category of set-valued presheaves and so can be given the standard category-with-families structure for such a category [9, Section 4]. However, in this case the structure is quite simple (if one is familiar with monoid actions): as we will see, one just uses a dependently-typed version of monoid action. We begin by recalling briefly the definition of category-with-families in order to fix notation; see [9] for more details and [1] for a more abstract, category-theoretic perspective.

▶ **Definition 3.1** (Category with families [5])**.** A *category with families* (*CwF*) is specified by a category $\mathcal{C}$ with a terminal object 1, together with the following structure:

- For each object $\Gamma \in \mathcal{C}$, a collection $\mathcal{C}(\Gamma)$, whose elements are called *families* over Γ.
- For each object $\Gamma \in \mathcal{C}$ and family $A \in \mathcal{C}(\Gamma)$, a collection $\mathcal{C}(\Gamma \vdash A)$ of *elements* of the family A over Γ.
- Operations for re-indexing families and elements along morphisms in $\mathcal{C}$

$$\frac{A \in \mathcal{C}(\Gamma) \qquad \gamma \in \mathcal{C}(\Gamma', \Gamma)}{A[\gamma] \in \mathcal{C}(\Gamma')} \qquad \frac{a \in \mathcal{C}(\Gamma \vdash A) \qquad \gamma \in \mathcal{C}(\Gamma', \Gamma)}{a[\gamma] \in \mathcal{C}(\Gamma' \vdash A[\gamma])}$$

satisfying

$$\begin{aligned} A[\mathrm{id}_\Gamma] &= A && (A \in \mathcal{C}(\Gamma)) \\ A[\gamma \circ \gamma'] &= A[\gamma][\gamma'] && (A \in \mathcal{C}(\Gamma), \gamma \in \mathcal{C}(\Gamma', \Gamma), \gamma' \in \mathcal{C}(\Gamma'', \Gamma')) \\ a[\mathrm{id}_\Gamma] &= a && (a \in \mathcal{C}(\Gamma \vdash A) \\ a[\gamma \circ \gamma'] &= a[\gamma][\gamma'] && (a \in \mathcal{C}(\Gamma \vdash A), \gamma \in \mathcal{C}(\Gamma', \Gamma), \gamma' \in \mathcal{C}(\Gamma'', \Gamma')) \end{aligned}$$

- For each family $A \in \mathcal{C}(\Gamma)$, a *comprehension* object $\Gamma.A \in \mathcal{C}$ equipped with a projection morphism $\mathrm{p} \in \mathcal{C}(\Gamma.A, \Gamma)$, a *generic element* $\mathrm{v} \in \mathcal{C}(\Gamma.A \vdash A[\mathrm{p}])$ and a *pairing operation*

$$\frac{\gamma \in \mathcal{C}(\Gamma', \Gamma) \qquad A \in \mathcal{C}(\Gamma) \qquad a \in \mathcal{C}(\Gamma' \vdash A[\gamma])}{\langle \gamma \,,\, a \rangle \in \mathcal{C}(\Gamma', \Gamma.A)}$$

satisfying

$$\begin{aligned} \mathrm{p} \circ \langle \gamma \,,\, a \rangle &= \gamma \\ \mathrm{v}[\langle \gamma \,,\, a \rangle] &= a \\ \langle \gamma \,,\, a \rangle \circ \gamma' &= \langle \gamma \circ \gamma' \,,\, a[\gamma'] \rangle \\ \langle \mathrm{p} \,,\, \mathrm{v} \rangle &= \mathrm{id}_{\Gamma.A} \end{aligned}$$

For each object $\Gamma \in \mathcal{C}$, one can make $\mathcal{C}(\Gamma)$ into a category by taking, for each $A, B \in \mathcal{C}(\Gamma)$, the set of morphisms $\mathcal{C}(\Gamma)(A, B)$ to be $\mathcal{C}(\Gamma.A \vdash B[\mathrm{p}])$ with identities given by generic elements and composition given by $c \circ b \triangleq c[\langle \mathrm{p} \,,\, b \rangle]$. Then the mapping $A \in \mathcal{C}(\Gamma) \mapsto \mathrm{p} \in \mathcal{C}(\Gamma.A, \Gamma)$ extends to a full and faithful functor to the slice category

$$\mathcal{C}(\Gamma) \to \mathcal{C}/\Gamma \tag{31}$$

$$A \xrightarrow{b} B \;\mapsto\; \begin{array}{ccc} \Gamma.A & \xrightarrow{\langle \mathrm{p}, b \rangle} & \Gamma.B \\ & {\scriptstyle \mathrm{p}} \searrow \quad \swarrow {\scriptstyle \mathrm{p}} & \\ & \Gamma & \end{array}$$

The re-indexing operations are mapped to pullback functors between slices, since for each $A \in \mathcal{C}(\Gamma)$ and $\gamma \in \mathcal{C}(\Gamma', \Gamma)$

$$\begin{array}{ccc} \Gamma'.A[\gamma] & \xrightarrow{\langle \gamma \circ \mathrm{p}, \mathrm{v} \rangle} & \Gamma.A \\ {\scriptstyle \mathrm{p}} \downarrow & \lrcorner & \downarrow {\scriptstyle \mathrm{p}} \\ \Gamma' & \xrightarrow[\gamma]{} & \Gamma \end{array} \tag{32}$$

is a pullback in $\mathcal{C}$; see [9, Proposition 3.9].

The contexts, types-in-context, terms-in-context and term-substitutions of Type Theory are interpreted in a CwF by its objects, families, elements and morphisms respectively; see [9, Section 3.5]. Furthermore, one can translate each type-forming construct to an equivalent structure within CwFs. For example:

▶ **Definition 3.2** (Σ- and Π-types in CwFs)**.** A Cwf $\mathcal{C}$ has

- Σ-*types* if there are operations

$$\frac{A \in \mathcal{C}(\Gamma) \qquad B \in \mathcal{C}(\Gamma.A)}{\Sigma\, A\, B \in \mathcal{C}(\Gamma)} \qquad \frac{a \in \mathcal{C}(\Gamma \vdash A) \qquad B \in \mathcal{C}(\Gamma.A) \qquad b \in \mathcal{C}(\Gamma \vdash B[\langle \mathrm{id}_\Gamma\, , a\rangle])}{\mathrm{pair}\, a\, b \in \mathcal{C}(\Gamma \vdash \Sigma\, A\, B)}$$

$$\frac{c \in \mathcal{C}(\Gamma \vdash \Sigma\, A\, B)}{\mathrm{fst}\, c \in \mathcal{C}(\Gamma \vdash A)} \qquad \frac{c \in \mathcal{C}(\Gamma \vdash \Sigma\, A\, B)}{\mathrm{snd}\, c \in \mathcal{C}(\Gamma \vdash B[\langle \mathrm{id}_\Gamma\, , \mathrm{fst}\, c\rangle])}$$

satisfying

$$\begin{aligned}
(\Sigma\, A\, B)[\gamma] &= \Sigma(A[\gamma])(B[\langle \gamma \circ \mathrm{p}\, , \mathrm{v}\rangle]) \\
(\mathrm{pair}\, a\, b)[\gamma] &= \mathrm{pair}\, (a[\gamma])\, (b[\gamma]) \\
(\mathrm{fst}\, c)[\gamma] &= \mathrm{fst}(c[\gamma]) \\
(\mathrm{snd}\, c)[\gamma] &= \mathrm{snd}(c[\gamma]) \\
\mathrm{fst}(\mathrm{pair}\, a\, b) &= a \\
\mathrm{snd}(\mathrm{pair}\, a\, b) &= b \\
\mathrm{pair}\, (\mathrm{fst}\, c)\, (\mathrm{snd}\, c) &= c
\end{aligned}$$

- Π-*types* if there are operations

$$\frac{A \in \mathcal{C}(\Gamma) \qquad B \in \mathcal{C}(\Gamma.A)}{\Pi\, A\, B \in \mathcal{C}(\Gamma)} \qquad \frac{b \in \mathcal{C}(\Gamma.A \vdash B)}{\mathrm{lam}\, b \in \mathcal{C}(\Gamma \vdash \Pi\, A\, B)} \qquad \frac{\begin{array}{c} c \in \mathcal{C}(\Gamma \vdash \Pi\, A\, B) \\ a \in \mathcal{C}(\Gamma \vdash A) \end{array}}{\mathrm{app}\, c\, a \in \mathcal{C}(\Gamma \vdash B[\langle \mathrm{id}_\Gamma\, , a\rangle])}$$

satisfying

$$\begin{aligned}
(\Pi\, A\, B)[\gamma] &= \Pi(A[\gamma])(B[\langle \gamma \circ \mathrm{p}\, , \mathrm{v}\rangle]) \\
(\mathrm{lam}\, b)[\gamma] &= \mathrm{lam}\, b[\langle \gamma \circ \mathrm{p}\, , \mathrm{v}\rangle] \\
(\mathrm{app}\, c\, a)[\gamma] &= \mathrm{app}\, (c[\gamma])\, (a[\gamma]) \\
\mathrm{app}\, (\mathrm{lam}\, b)\, a &= b[\langle \mathrm{id}_\Gamma\, , a\rangle] \\
\mathrm{lam}(\mathrm{app}\, (c[\mathrm{p}])\, \mathrm{v}) &= c
\end{aligned}$$

▶ Remark 3.3. If $\mathcal{C}$ is a locally cartesian closed category, it is always possible to find a CwF with the same underlying category $\mathcal{C}$ for which the functors in (31) are not only full and faithful, but also essentially surjective; see [13, 1]. In that case each category of families $\mathcal{C}(\Gamma)$ is equivalent to the slice category $\mathcal{C}/\Gamma$ and the CwF structure is just providing an equivalent version of the traditional use of slice categories to model families of objects in category theory [16] – one in which pullback strictly commutes with composition and hence correctly models properties of substitution in type theory. This applies to the categories we consider in this paper, $[\mathbf{C}, \mathbf{Set}]$, $\mathbf{Set}^{\mathbf{M}}$ and $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$, since they are all toposes and hence in particular locally cartesian closed. However, in these cases it not necessary to apply a general construction as in [13, 1], since there are natural and useful notions of 'family of presheaves' and 'family of $\mathbf{M}$-sets' equivalent to the use of slice categories. Such families are used in [2] for the category of cubical sets; and we describe analogues for $\mathbf{Set}^{\mathbf{M}}$ and $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ in the next two sections. Note that the equivalence $I^* : [\mathbf{C}, \mathbf{Set}] \simeq \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ from Theorem 2.9 gives an equivalence $[\mathbf{C}, \mathbf{Set}]/C \simeq \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}/I^*C$ for each cubical set C; and therefore the category of families over C, being equivalent to $[\mathbf{C}, \mathbf{Set}]/C$ and hence to $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}/I^*C$, is also equivalent to the category of families for I^*C in the CwF described in Section 3.2 for the case $\mathbf{M} = \mathbf{Cb}$.

3.1 CwF structure of $\mathrm{Set}^{\mathbf{M}}$

Let $\mathbf{M}$ be an arbitrary monoid.

- *Families* $\mathbf{Set}^{\mathbf{M}}(\Gamma)$ over an object $\Gamma \in \mathbf{Set}^{\mathbf{M}}$ consist of Γ-indexed families of sets $A = (A_d \mid d \in \Gamma)$ equipped with a 'dependently-typed $\mathbf{M}$-action', that is, a family of functions
$$\_ \cdot \_ \in \prod_{\sigma \in \mathbf{M}} \prod_{a \in A_d} A_{\sigma \cdot d} \qquad (d \in \Gamma)$$
satisfying $\iota \cdot a = a \in A_d (= A_{\iota \cdot d})$ and $\sigma' \cdot (\sigma \cdot a) = \sigma'\sigma \cdot a \in A_{\sigma'\sigma \cdot d} (= A_{\sigma' \cdot (\sigma \cdot d)})$.
- *Elements* $\mathbf{Set}^{\mathbf{M}}(\Gamma \vdash A)$ of a family $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ consist of dependently-typed functions $f \in \prod_{d \in \Gamma} A_d$ that preserve the $\mathbf{M}$-action, in the sense that $\sigma \cdot (f\, d) = f(\sigma \cdot d) \in A_{\sigma \cdot d}$.
- *Re-indexing* of a family $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ along $\gamma \in \mathbf{Set}^{\mathbf{M}}(\Gamma', \Gamma)$ is the family $A[\gamma] \triangleq (A_{\gamma\, d'} \mid d' \in \Gamma')$ with dependently-typed $\mathbf{M}$-action: $\sigma \in \mathbf{M}, a \in A[\gamma]_{d'} = A_{\gamma\, d'} \mapsto \sigma \cdot a \in A_{\sigma \cdot (\gamma\, d')} = A_{\gamma(\sigma \cdot d')} = A[\gamma]_{\sigma \cdot d'}$. The re-indexing of an element $f \in \mathbf{Set}^{\mathbf{M}}(\Gamma \vdash A)$ along $\gamma \in \mathbf{Set}^{\mathbf{M}}(\Gamma', \Gamma)$ is the element $f[\gamma] \in \mathbf{Set}^{\mathbf{M}}(\Gamma' \vdash A)$, where $f[\gamma]\, d' = f(\gamma\, d')$.
- *Comprehension* for the CwF $\mathbf{Set}^{\mathbf{M}}$ is created by that for $\mathbf{Set}$. Thus given $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$, the comprehension object $\Gamma.A \in \mathbf{Set}^{\mathbf{M}}$ is given by the dependent product of sets
$$\Gamma.A \triangleq \textstyle\sum_{d \in \Gamma} A_d \quad \text{equipped with the } \mathbf{M}\text{-action} \quad \sigma \cdot (d, a) \triangleq (\sigma \cdot d, \sigma \cdot a) \tag{33}$$
First projection yields a morphism $\mathrm{p} \in \mathbf{Set}^{\mathbf{M}}(\Gamma.A, \Gamma)$ and the generic element $\mathrm{v} \in \mathbf{Set}^{\mathbf{M}}(\Gamma.A \vdash A[\mathrm{p}])$ is given by second projection: $\mathrm{v}(d, a) \triangleq a \in A_d = A[\mathrm{p}]_{(d,a)}$. The pairing operation is
$$\frac{\gamma \in \mathbf{Set}^{\mathbf{M}}(\Gamma', \Gamma) \qquad f \in \mathbf{Set}^{\mathbf{M}}(\Gamma' \vdash A[\gamma])}{\langle \gamma\, , f \rangle \in \mathbf{Set}^{\mathbf{M}}(\Gamma', \Gamma.A)} \qquad \langle \gamma\, , f \rangle\, d' \triangleq (\gamma\, d', f\, d') \qquad (d' \in \Gamma')$$

These operations satisfy the equations required for a CwF (Definition 3.1). In this case the functors (31) are equivalences: any object $\gamma : \Gamma' \to \Gamma$ of the slice category $\mathbf{Set}^{\mathbf{M}}/\Gamma$ is isomorphic to $\mathrm{p} : \Gamma.A \to \Gamma$ for some family $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$, namely $A_d \triangleq \{d' \in \Gamma' \mid \gamma\, d' = d\}$ with dependently-typed action given by the $\mathbf{M}$-action of Γ'. Since $\mathbf{Set}^{\mathbf{M}}$ is a topos (being a presheaf category), it is in particular locally cartesian closed. One can use the equivalences $\mathbf{Set}^{\mathbf{M}}(\Gamma) \simeq \mathbf{Set}^{\mathbf{M}}/\Gamma$ to transfer this local cartesian closed structure to operations in the CwF $\mathbf{Set}^{\mathbf{M}}$ for modelling Σ- and Π-types (Definition 3.2). Given families $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ and $B \in \mathbf{Set}^{\mathbf{M}}(\Gamma.A)$, then $\Sigma\, A\, B \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ is given by the dependent product of sets
$$(\Sigma\, A\, B)_d \triangleq \textstyle\sum_{a \in A_d} B_{(d,a)} \quad \text{equipped with } \mathbf{M}\text{-action} \quad \sigma \cdot (a, b) \triangleq (\sigma \cdot a, \sigma \cdot b) \tag{34}$$
with $\mathrm{pair}\, a\, b$, $\mathrm{fst}\, c$ and $\mathrm{snd}\, c$ as for $\mathbf{Set}$. However, $\Pi\, A\, B \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ is more complicated:
$$(\Pi\, A\, B)_d \triangleq \{f \in \textstyle\prod_{\sigma' \in \mathbf{M}} \prod_{a \in A_{\sigma' \cdot d}} B_{(\sigma' \cdot d, a)} \mid (\forall \sigma, \sigma' \in \mathbf{M})(\forall a \in A_{\sigma' \cdot d}) \\ \sigma \cdot (f\, \sigma'\, a) = f\, (\sigma\sigma')\, (\sigma \cdot a) \in B_{(\sigma\sigma' \cdot d, \sigma \cdot a)}\} \qquad (d \in \Gamma) \tag{35}$$
with $\mathbf{M}$-action:
$$\frac{\sigma \in \mathbf{M} \qquad f \in (\Pi\, A\, B)_d}{\sigma \cdot f \in (\Pi\, A\, B)_{\sigma \cdot d}} \qquad \sigma \cdot f \triangleq \lambda \sigma' \in \mathbf{M}.\lambda a \in A_{\sigma'\sigma \cdot d}.\, f\, (\sigma'\sigma)\, a$$
Application is given by
$$\frac{g \in \mathbf{Set}^{\mathbf{M}}(\Gamma \vdash \Pi\, A\, B) \qquad h \in \mathbf{Set}^{\mathbf{M}}(\Gamma \vdash A)}{\mathrm{app}\, g\, h \in \mathbf{Set}^{\mathbf{M}}(\Gamma \vdash B[\langle \mathrm{id}\, , h \rangle])} \qquad \mathrm{app}\, g\, h\, d \triangleq g\, d\, \iota\, (h\, d) \qquad (d \in \Gamma) \tag{36}$$
and currying by
$$\frac{k \in \mathbf{Set}^{\mathbf{M}}(\Gamma.A \vdash B)}{\mathrm{lam}\, k \in \mathbf{Set}^{\mathbf{M}}(\Gamma \vdash \Pi\, A\, B)} \qquad \mathrm{lam}\, k\, d \triangleq \lambda \sigma' \in \mathbf{M}.\lambda a \in A_{\sigma' \cdot d} k\, (\sigma' \cdot d, a) \qquad (d \in \Gamma) \tag{37}$$

3.2 CwF structure of $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$

Now let $\mathbf{M}$ be a monoid of substitutions (Definition 2.1)

▶ **Lemma 3.4.** *The full subcategory* $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ *is closed under taking finite limits in* $\mathbf{Set}^{\mathbf{M}}$.

Proof. Just note that for each element $(d_1, \ldots, d_n)$ of a finite limit, since the action of $\mathbf{M}$ on the finite limit is componentwise, if each component d_i is supported by X_i, the the whole element is supported by $X_1 \cup \cdots \cup X_n$. ◀

▶ **Lemma 3.5.** *Given* $\Gamma \in \mathbf{Set}^{\mathbf{M}}$, *define*

$$\Gamma_{\mathrm{fs}} \triangleq \{d \in \Gamma \mid d \text{ is supported by some finite subset } X \subseteq_{\mathrm{fin}} \mathbb{D}\}$$

Then $\Gamma \mapsto \Gamma_{\mathrm{fs}}$ *is the object part of a right adjoint to the inclusion functor* $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}} \hookrightarrow \mathbf{Set}^{\mathbf{M}}$.

Proof. First note that by part 2 of Corollary 2.5, Γ_{fs} is closed under the $\mathbf{M}$-action on Γ and hence gives an object in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$. For the right adjointness we just have to see that given $\Gamma' \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$, any morphism $\gamma \in \mathbf{Set}^{\mathbf{M}}(\Gamma', \Gamma)$ factors (necessarily uniquely) through the inclusion $\Gamma_{\mathrm{fs}} \hookrightarrow \Gamma$. But if $d' \in \Gamma'$ is supported by $X \subseteq_{\mathrm{fin}} \mathbb{D}$, then by part 1 of Corollary 2.5, $\gamma\, d' \in \Gamma$ is also supported by X. ◀

▶ Remark 3.6. Combining Lemmas 3.4 and 3.5, we have that if $\mathbf{M}$ is a monoid of substitutions, then $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ is a topos and there is a geometric surjection $\mathbf{Set}^{\mathbf{M}} \to \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ whose direct image part is the right adjoint functor $(\_)_{\mathrm{fs}} : \mathbf{Set}^{\mathbf{M}} \to \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ (see [11, Proposition 4.15(ii)], for example).

The CwF structure on $\mathbf{Set}^{\mathbf{M}}$ given above restricts to one for $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ when $\mathbf{M}$ is a monoid of substitutions. For each $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ we define:

- *Families* $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ are families of $\mathbf{M}$-sets $A \in \mathbf{Set}^{\mathbf{M}}(\Gamma)$ for which the comprehension object (33) is in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$. This amounts to requiring that for each $d \in \Gamma$, every $a \in A_d$ possesses a finite support with respect to the dependently-typed $\mathbf{M}$-action, that is, a finite subset $X \subseteq_{\mathrm{fin}} \mathbb{D}$ satisfying $\mathrm{supp}\, d \subseteq X$ and $(\forall \sigma, \sigma' \in \mathbf{M})((\forall \mathsf{x} \in X)\ \sigma\, \mathsf{x} = \sigma'\, \mathsf{x}) \Rightarrow \sigma \cdot a = \sigma' \cdot a$. (The condition $\mathrm{supp}\, d \subseteq X$, i.e. X supports d, is important since with it, when $(\forall \mathsf{x} \in X)\ \sigma\, \mathsf{x} = \sigma'\, \mathsf{x}$ holds, it makes sense to compare $\sigma \cdot a$ and $\sigma' \cdot a$ for equality, because we have $\sigma \cdot a \in A_{\sigma \cdot d} = A_{\sigma' \cdot d} \ni \sigma' \cdot a$.) Note that the functor from Lemma 3.5 extends to a fibre-wise version:

$$\begin{aligned} &\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}, A \in \mathbf{Set}^{\mathbf{M}}(\Gamma) \ \mapsto\ A_{\mathrm{fs}} \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma) \\ &(A_{\mathrm{fs}})_d \triangleq \{a \in A_d \mid (d, a) \in (\Gamma.A)_{\mathrm{fs}}\} \qquad (d \in \Gamma) \end{aligned} \tag{38}$$

- *Elements* $f \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma \vdash A)$ are the same as in $\mathbf{Set}^{\mathbf{M}}$, namely dependently-typed functions $f \in \prod_{d \in \Gamma} A_d$ that preserve the $\mathbf{M}$-action.
- *Re-indexing* is the same as in $\mathbf{Set}^{\mathbf{M}}$, since if X supports $(\gamma\, d', a)$ in $\Gamma.A$, then it supports (d', a) in $\Gamma'.A[\gamma]$.
- *Comprehension objects* are as in (33), since by definition $\Gamma.A$ is in the subcategory $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ when $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$.

▶ Remark 3.7. We noted above that the functors (31) give equivalences when $\mathcal{C} = \mathbf{Set}^{\mathbf{M}}$. Because of the definition of families in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ (and the fact that the objects of $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ are closed under isomorphisms in $\mathbf{Set}^{\mathbf{M}}$), it follows that (31) is also an equivalence when $\mathcal{C} = \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$.

Σ-types in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ are as in (34), since $(a, b) \in (\Sigma\, A\, B)_d$ is supported by any $X \supseteq \mathrm{supp}(d, a)$ that supports $b \in B_{(d,a)}$.

Π-types in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ are obtained by applying the functor (38) to (35). Thus for each $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$, $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ and $B \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma.A)$ we define $\Pi_{\mathrm{fs}}\, A\, B \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ by

$$(\Pi_{\mathrm{fs}}\, A\, B)_d \triangleq ((\Pi\, A\, B)_{\mathrm{fs}})_d \qquad (d \in \Gamma) \tag{39}$$

and one can check that the application (36) and currying (37) operations preserve the finite support property. Combining (35) with (39) in the case that $\Gamma = 1$, we recover the following description of exponentials in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ that will be useful later.

▶ **Lemma 3.8** (Exponentials in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$). *Given $\Gamma, \Delta \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$, their exponential Δ^Γ is given by the set $(\Gamma \to_{\mathbf{M}} \Delta)_{\mathrm{fs}}$ of finitely supported elements of*

$$\Gamma \to_{\mathbf{M}} \Delta \;\triangleq\; \{f \in \mathbf{Set}(\mathbf{M} \times \Gamma, \Delta) \mid (\forall \sigma, \sigma' \in \mathbf{M})(\forall d \in \Gamma)\ \sigma \cdot f(\sigma', d) = f(\sigma\sigma', \sigma \cdot d)\}$$

where the $\mathbf{M}$-action on $\Gamma \to_{\mathbf{M}} \Delta$ is

$$\sigma \cdot f \;\triangleq\; \lambda(\sigma', d) \in \mathbf{M} \times \Gamma.\, f(\sigma'\sigma, d)$$

The evaluation morphism $\mathrm{ev} \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Delta^\Gamma \times \Gamma, \Delta)$ is given by $\mathrm{ev}(f, d) = f(\iota, d)$; and the currying of $\gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma' \times \Gamma, \Delta)$ is $\mathrm{cur}\,\gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma', \Delta^\Gamma)$, where $\mathrm{cur}\,\gamma\, d' = \lambda(\sigma, d) \in \mathbf{M} \times \Gamma.\, \gamma(\sigma \cdot d', d)$. ◀

3.3 Hofmann-Streicher universes

Hofmann and Streicher [10] describe a way of lifting a Grothendieck universe in **Set** to a type-theoretic universe in any presheaf category. This is used by Bezem, Coquand and Huber [2] to construct a universe within the category $[\mathbf{C}, \mathbf{Set}]$ of cubical sets. We give the construction for the case when the presheaf category is $\mathbf{Set}^{\mathbf{M}}$ for a monoid $\mathbf{M}$ and then apply the coreflection $(\_)_{\mathrm{fs}} : \mathbf{Set}^{\mathbf{M}} \to \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ from Lemma 3.5 when $\mathbf{M}$ is a monoid of substitutions.

Let $\mathcal{U}$ be a Grothendieck universe (see [19], for example) containing $\mathbb{D}$ and hence also $\mathbf{M}$. We lift $\mathcal{U}$ to an object U of $\mathbf{Set}^{\mathbf{M}}$ whose underlying set consists of certain pairs (F, act) where F is a function from $\mathbf{M}$ to $\mathcal{U}$ and $act \in \prod_{\sigma \in \mathbf{M}} \prod_{\sigma' \in \mathbf{M}} (F(\sigma'\sigma))^{F\,\sigma}$. Thus F is an $\mathbf{M}$-indexed family of sets $F\,\sigma \in \mathcal{U}$ ($\sigma \in \mathbf{M}$) and act maps $\sigma, \sigma' \in \mathbf{M}$ to a function $act\,\sigma\,\sigma' : F\,\sigma \to F(\sigma'\sigma)$. We use the following notation for act:

$$\sigma' \cdot a \;\triangleq\; act\,\sigma\,\sigma'\,a \in F(\sigma'\sigma) \qquad (\sigma' \in \mathbf{M}, a \in F\,\sigma) \tag{40}$$

and refer to (F, act) via F. For it to be in U we require act to be a dependently typed $\mathbf{M}$-action (cf. Section 3.1), in the sense that if $a \in F\,\sigma$, then

$$\iota \cdot a = a \in F\,\sigma = F(\iota\sigma) \tag{41}$$

$$\sigma'' \cdot (\sigma' \cdot a) = (\sigma''\sigma') \cdot a \in F((\sigma''\sigma')\sigma) = F(\sigma''(\sigma'\sigma)) \tag{42}$$

If $F \in U$ and $\sigma \in \mathbf{M}$, then we get $\sigma \cdot F \in U$ by defining

$$(\sigma \cdot F)\,\sigma' \triangleq F(\sigma'\sigma) \tag{43}$$

with dependently typed $\mathbf{M}$-action on $\sigma \cdot F$ given by the one for F. (This makes sense, since if $a \in (\sigma \cdot F)\,\sigma' = F(\sigma'\sigma)$, then $\sigma'' \cdot a \in F(\sigma''(\sigma'\sigma)) = (\sigma \cdot F)(\sigma''\sigma')$.) These definitions make U into an object in $\mathbf{Set}^{\mathbf{M}}$, since one can easily check that $\iota \cdot F = F$ and $\sigma' \cdot (\sigma \cdot F) = (\sigma'\sigma) \cdot F$.

▶ **Definition 3.9 (Hofmann-Streicher lifting for $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$).** Let $\mathbf{M}$ be a monoid of substitutions and let $U \in \mathbf{Set}^{\mathbf{M}}$ be the $\mathbf{M}$-set derived from a Grothendieck universe $\mathcal{U} \in \mathbf{Set}$ as above. Let $E \in \mathbf{Set}^{\mathbf{M}}(U)$ be the family mapping each $F \in U$ to

$$E_F \triangleq F\,\iota \quad \text{with dependently-typed } \mathbf{M}\text{-action given by (40)}$$

(Note that this makes sense, because if $a \in E_F = F\,\iota$, then $\sigma \cdot a \in F(\sigma\iota) = F(\iota\sigma) = (\sigma \cdot F)\,\iota = E_{\sigma\cdot F}$.) Applying the functor $(\_)_{\mathrm{fs}} : \mathbf{Set}^{\mathbf{M}} \to \mathbf{Sub}$ from Lemma 3.5 to the projection morphism $\mathrm{p} : U.E \to U$ we get a morphism $\mathrm{p} : (U.E)_{\mathrm{fs}} \to U_{\mathrm{fs}}$ in $\mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ and (by Remark 3.7) a corresponding family $E_{\mathrm{fs}} \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(U_{\mathrm{fs}})$, where

$$(E_{\mathrm{fs}})_F \triangleq \{a \in F\,\iota \mid (F, a) \in (U.E)_{\mathrm{fs}}\} \qquad (F \in U_{\mathrm{fs}}) \tag{44}$$

Note that if $F \in U_{\mathrm{fs}}$, then $F\,\iota \in \mathcal{U}$ and hence $(E_{\mathrm{fs}})_F \in \mathcal{U}$. In general we say that a family $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ *has fibres in* $\mathcal{U}$ if $A_d \in \mathcal{U}$ for all $d \in \Gamma$. The family (44) not only has fibres in $\mathcal{U}$, but is weakly universal among such families, in the following sense.

▶ **Theorem 3.10.** *Let* $\mathbf{M}$ *be a monoid of substitutions and* $E_{\mathrm{fs}} \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(U_{\mathrm{fs}})$ *be the Hofmann-Streicher universe in the CwF of finitely supported* $\mathbf{M}$*-sets derived from a Grothendieck universe* $\mathcal{U} \in \mathbf{Set}$*. Then for each* $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$ *and family* $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ *with fibres in* $\mathcal{U}$*, there is a morphism* $\ulcorner A \urcorner \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma, U_{\mathrm{fs}})$ *with* $A = E_{\mathrm{fs}}[\ulcorner A \urcorner]$.

Proof. For each $d \in \Gamma$ and $\sigma \in \mathbf{M}$ define

$$\ulcorner A \urcorner\, d\, \sigma \triangleq A_{\sigma\cdot d} \in \mathcal{U} \tag{45}$$

If $\sigma' \in \mathbf{M}$ and $a \in \ulcorner A \urcorner\, d\, \sigma = A_{\sigma\cdot d}$, then the dependently-typed $\mathbf{M}$-action on $A \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma)$ gives us $\sigma' \cdot a \in A_{\sigma'\cdot(\sigma\cdot d)} = \ulcorner A \urcorner\, d\, (\sigma'\sigma)$, satisfying (41) and (42). So for each $d \in \Gamma$, we get $\ulcorner A \urcorner\, d \in U$. Furthermore

$$\begin{aligned}
(\sigma' \cdot (\ulcorner A \urcorner\, d))\,\sigma &= (\ulcorner A \urcorner\, d)(\sigma\sigma') && \text{by (43)}\\
&= A_{\sigma\sigma'\cdot d} && \text{by (45)}\\
&= A_{\sigma\cdot(\sigma'\cdot d)} &&\\
&= \ulcorner A \urcorner\, (\sigma' \cdot d)\,\sigma && \text{by (45) again}
\end{aligned}$$

so that $\ulcorner A \urcorner \in \mathbf{Set}^{\mathbf{M}}(\Gamma, U)$. Since $\Gamma \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}$, if follows from Lemma 3.5 that $\ulcorner A \urcorner$ factors through $U_{\mathrm{fs}} \hookrightarrow U$ to give $\ulcorner A \urcorner \in \mathbf{Set}^{\mathbf{M}}_{\mathrm{fs}}(\Gamma, U_{\mathrm{fs}})$. Since it follows from this that $\mathrm{supp}(\ulcorner A \urcorner\, d) \subseteq \mathrm{supp}\, d$, if $a \in A_d$ is supported by $X \supseteq \mathrm{supp}\, d$, then X also supports $(\ulcorner A \urcorner\, d, a)$ in $U.E$. Therefore by (44), for all $d \in \Gamma$ we have

$$A_d = \{a \in \ulcorner A \urcorner\, d\,\iota \mid (\ulcorner A \urcorner\, d, a) \in (U.E)_{\mathrm{fs}}\} = E_{\mathrm{fs}}[\ulcorner A \urcorner]_d$$

so that re-indexing the family E_{fs} along $\ulcorner A \urcorner$ gives $E_{\mathrm{fs}}[\ulcorner A \urcorner] = A$. ◀

4 Cubical sets with diagonals

In footnote 2 of [2] the authors say

> 'In a previous attempt, we have been considering the category of finite sets with maps $I \to J + 2$ (i.e. the Kleisli category for the monad $I + 2$). This category appears on pages 47–48 in Pursuing Stacks [8] as "in a sense, the smallest test category".'

Call this category $\mathbf{S}$. Thus $\mathbf{S}$ is like the category $\mathbf{C}$ from Section 1, but without the injectivity condition (1) on morphisms. In Section 2 we moved from the small category $\mathbf{C}$ to the submonoid $\mathbf{Cb}$ of the monoid $\mathbf{Sb}$ of all substitutions (Definition 2.1) and replaced cubical sets $[\mathbf{C}, \mathbf{Set}]$ by the equivalent category $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ of finitely supported $\mathbf{Cb}$-sets. If one starts from $\mathbf{S}$ rather than $\mathbf{C}$, then one gets the whole monoid of substitutions $\mathbf{Sb}$ and can consider the category $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ of finitely supported $\mathbf{Sb}$-sets.

▶ **Theorem 4.1.** *The categories* $[\mathbf{S}, \mathbf{Set}]$ *and* $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ *are equivalent.*

Proof. One can check that the proof method of Theorem 2.9 still goes through when one replaces the category $\mathbf{C}$ by $\mathbf{S}$ and the monoid $\mathbf{Cb}$ by $\mathbf{Sb}$. Indeed the proof is easier, because the 'homogeneity' property (the analogue of Lemma 2.8) needed for the fullness and essential surjectivity of the functor $I^* : [\mathbf{S}, \mathbf{Set}] \to \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ is trivial: for each $s \in \mathbf{S}(X, Y)$ we get a substitution $\sigma \in \mathbf{Sb}$ that agrees with s on X simply by defining

$$\sigma\,\mathsf{x} \triangleq \begin{cases} s\,\mathsf{x} & \text{if } \mathsf{x} \in X \\ \mathsf{x} & \text{otherwise.} \end{cases}$$

◀

One advantage of $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ over $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$ stems from the following theorem. Regarding each $\Gamma \in \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ as a nominal set as in Section 2.1, we can make the nominal set $[\mathbb{D}]\Gamma$ of name abstractions into an object of $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ via an $\mathbf{Sb}$-action satisfying

$$\mathsf{x} \# \sigma \;\Rightarrow\; \sigma \cdot \langle \mathsf{x} \rangle d = \langle \mathsf{x} \rangle (\sigma \cdot d) \qquad (\mathsf{x} \in \mathbb{D}, \sigma \in \mathbf{Sb}, d \in \Gamma) \tag{46}$$

where we regard $\mathbf{Sb}$ as a nominal set, and hence make sense of the condition $\mathsf{x} \# \sigma$, via the conjugation action of permutations: $\pi \cdot \sigma \triangleq \pi\sigma\pi^{-1}$. (The support of σ with respect to this action is $\mathrm{Dom}\,\sigma \cup \bigcup_{\mathsf{x} \in \mathrm{Dom}\,\sigma} \mathrm{supp}(\sigma\,\mathsf{x})$.) Thus the action of σ is well-defined by sending an element $\langle \mathsf{x} \rangle d \in [\mathbb{D}]\Gamma$ to $\langle \mathsf{y} \rangle (\sigma(\mathsf{x}\ \mathsf{y}) \cdot d)$, where y is some (or indeed, any) direction satisfying $\mathsf{y} \# (\mathsf{x}, \sigma, d)$; cf. [15, Theorem 9.18].

▶ **Theorem 4.2.** $[\mathbb{D}]\Gamma$ *is isomorphic in the category* $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ *to the exponential* Γ^{I} *of* Γ *by the interval object* I *from Example 2.3.*

Proof. Recall the definition of $[\mathbb{D}]\Gamma$ in terms of the equivalence relation $\approx_\alpha$ (26). If $(\mathsf{x}, d) \approx_\alpha (\mathsf{x}', d')$, then picking any $\mathsf{y} \# (\mathsf{x}, d, \mathsf{x}', d')$ we have $(\mathsf{y}\ \mathsf{x}) \cdot d = (\mathsf{y}\ \mathsf{x}') \cdot d' \in \Gamma$. Since for any $i \in \mathrm{I}$, the substitutions $(i/\mathsf{y})(\mathsf{y}\ \mathsf{x})$ and (i/x) agree on $\mathrm{supp}\, d$, we have $(i/\mathsf{x}) \cdot d = (i/\mathsf{y})(\mathsf{y}\ \mathsf{x}) \cdot d$; similarly $(i/\mathsf{x}') \cdot d' = (i/\mathsf{y})(\mathsf{y}\ \mathsf{x}') \cdot d'$. Therefore $(i/\mathsf{x}) \cdot d = (i/\mathsf{x}') \cdot d'$. So there is a well-defined function $\mathrm{ev} : [\mathbb{D}]\Gamma \times \mathrm{I} \to \Gamma$ satisfying

$$\mathrm{ev}(\langle \mathsf{x} \rangle d, i) = (i/\mathsf{x}) \cdot d \qquad (\mathsf{x} \in \mathbb{D}, d \in \Gamma, i \in \mathrm{I}) \tag{47}$$

(Note that since $\mathbf{Cb}$ does not contain the substitution (i/x) when $i \in \mathbb{D} - \{\mathsf{x}\}$, it would not be possible to make this definition in the category $\mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$.)

When $\mathsf{x} \# \sigma$, we have $\sigma(i/\mathsf{x}) = (\sigma\, i/\mathsf{x})\sigma \in \mathbf{Sb}$ and hence $\sigma \cdot \mathrm{ev}(\langle \mathsf{x} \rangle d, i) = \mathrm{ev}(\sigma \cdot \langle \mathsf{x} \rangle d, \sigma \cdot i)$. So ev is a morphism in $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}([\mathbb{D}]\Gamma \times \mathrm{I}, \Gamma)$. We verify that it has the universal property required for the exponential. Given $\gamma \in \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}(\Gamma' \times \mathrm{I}, \Gamma)$ we get a well-defined function $\mathrm{cur}\,\gamma : \Gamma' \to [\mathbb{D}]\Gamma$

$$\mathrm{cur}\,\gamma\, d' \;\triangleq\; \langle \mathsf{x} \rangle \gamma(d', \mathsf{x}) \quad \text{where } \mathsf{x} \# d' \tag{48}$$

This satisfies $\sigma \cdot (\operatorname{cur}\gamma\, d') = \operatorname{cur}\gamma\,(\sigma \cdot d')$ and hence gives a morphism $\operatorname{cur}\gamma \in \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}(\Gamma', [\mathbb{D}]\Gamma)$. Note that

$$\begin{aligned}
\operatorname{ev}(\operatorname{cur}\gamma\, d', i) &= \operatorname{ev}(\langle \mathsf{x}\rangle \gamma(d', \mathsf{x}), i) && \text{where } \mathsf{x} \# d' \\
&= (i/\mathsf{x}) \cdot \gamma(d', \mathsf{x}) && \text{by (47)} \\
&= \gamma((i/\mathsf{x}) \cdot d', (i/\mathsf{x}) \cdot \mathsf{x}) && \text{since } \gamma \text{ is a morphisms in } \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}} \\
&= \gamma(d', i) && \text{since } \mathsf{x} \# d'
\end{aligned}$$

so that $\operatorname{ev} \circ (\operatorname{cur}\gamma \times \mathrm{id}_{\mathrm{I}}) = \gamma$. The uniqueness of $\operatorname{cur}\gamma$ with this property follows from an η-rule for elements of $[\mathbb{D}]\Gamma$:

$$(\forall p \in [\mathbb{D}]\Gamma)(\forall \mathsf{x} \in \mathbb{D})\ \mathsf{x} \# p \ \Rightarrow\ p = \langle \mathsf{x}\rangle \operatorname{ev}(p, \mathsf{x}) \tag{49}$$

which in turn follows the fact that for any $\langle \mathsf{x}\rangle d \in [\mathbb{D}]\Gamma$ and $\mathsf{y} \# (\mathsf{x}, d)$ it is the case that $\langle \mathsf{x}\rangle d = \langle \mathsf{y}\rangle((\mathsf{y}\ \mathsf{x}) \cdot d) = \langle \mathsf{y}\rangle((\mathsf{y}/\mathsf{x}) \cdot d)$. ◀

Iterating the theorem, we get that the exponential Γ^{I^n} (the object of n-cubes in Γ) is isomorphic to $[\mathbb{D}]^{(n)}\Gamma$, where

$$\begin{aligned}
[\mathbb{D}]^{(0)}\Gamma &\triangleq \Gamma \\
[\mathbb{D}]^{(n+1)}\Gamma &\triangleq [\mathbb{D}]([\mathbb{D}]^{(n)}\Gamma)
\end{aligned} \tag{50}$$

Note that $[\mathbb{D}]^{(n)}\Gamma \in \mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ is the nominal set of n-ary name abstractions $\langle \mathsf{x}_1, \ldots, \mathsf{x}_n\rangle d$ (with $\mathsf{x}_1, \ldots, \mathsf{x}_n$ mutually distinct directions) equipped with the **Sb**-action satisfying the evident generalisation of (46) to n-ary name abstractions.

One may think of objects of $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ as cubical sets 'with diagonals', because (unlike the case for $[\mathbf{C}, \mathbf{Set}] \simeq \mathbf{Set}^{\mathbf{Cb}}_{\mathrm{fs}}$) each square $\langle \mathsf{x}, \mathsf{y}\rangle d \in [\mathbb{D}]^{(2)}\Gamma$ contains a diagonal path $\langle \mathsf{z}\rangle(\mathsf{z}/\mathsf{x})(\mathsf{z}/\mathsf{y}) \cdot d \in [\mathbb{D}]\Gamma$. Of course, under the isomorphism in the above theorem, diagonalization $[\mathbb{D}]^{(2)}\Gamma \to [\mathbb{D}]\Gamma$ corresponds to the morphism $\Gamma^{\mathrm{I}^2} \to \Gamma^{\mathrm{I}}$ given by precomposing with the diagonal $\langle \mathrm{id}_{\mathrm{I}}\,, \mathrm{id}_{\mathrm{I}}\rangle : \mathrm{I} \to \mathrm{I} \times \mathrm{I}$.

▶ Remark. Gabbay and Hofmann [7] prove the analogue of Theorem 4.2 for their category of 'nominal renaming sets'. This category is like **01-Nom** except that it uses nominal sets equipped with name-for-name substitutions, rather than 01-for-name substitutions. They also have an analogue of Theorem 2.13: an equivalence between the category of nominal renaming sets and a sheaf subcategory the presheaf category $[\mathbf{F}, \mathbf{Set}]$, where $\mathbf{F}$ is the small category whose objects are finite subsets of $\mathbb{D}$ and whose morphisms are all functions between such subsets; see [7, Theorem 38].

5 Conclusion

We have shown how to reformulate cubical sets, originally given as presheaves, in terms of sets whose elements are finitely supported with respect to a given action of a monoid of name substitutions. Because of the equivalences we have established (Theorems 2.9, 2.13 and 4.1), there is no difference in the category-theoretic properties of the two formulations. However, the approach using monoids of name substitutions leads to a relatively simple notion of *family* of cubical sets (Section 3) and allows access to the well-developed nominal sets notions of *freshness* to calculate with degeneracy of cubes and *name abstraction* to calculate with paths (proofs of equality); see the implementation of Kan cubical sets [4].

We saw that in the category $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$, paths are arbitrary functions from an interval object (Theorem 4.2). Coquand [3] has noted that this property can enable simpler formulations of

the Kan filling condition, simpler proofs of closure of Kan complete families under taking Π-types, and more natural realizers for operations like the elimination rule for the circle. So there may be a sub-CwF of $\mathbf{Set}^{\mathbf{Sb}}_{\mathrm{fs}}$ consisting of families satisfying some Kan-filling condition which yields a technically simpler model of univalent foundations than the one in [2]. Of course, to be computationally useful, such a model has to exist in a constructive meta-theory. We leave this for future investigation.

Acknowledgements. The author wishes to thank Sam Staton, Thierry Coquand, Peter Aczel, the anonymous referees and members of the 'cubical seminar' at the Institut Henri Poincaré thematic trimester on *Semantics of proofs and certified mathematics* for discussion and comments. He is very grateful to the organizers of TYPES 2014 for inviting him to speak at the conference.

References

1 S. Awodey. Natural models of homotopy type theory. *ArXiv e-prints*, arXiv:1406.3219 [math.LO], June 2014.

2 M. Bezem, T. Coquand, and S. Huber. A model of type theory in cubical sets. In R. Matthes and A. 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.

3 T. Coquand. Variation on cubical sets. Preprint, 31 March, 2014.

4 Cubical. `github.com/simhu/cubical`.

5 Peter Dybjer. Internal type theory. In S. Berardi and M. Coppo, editors, *Types for Proofs and Programs*, volume 1158 of *Lecture Notes in Computer Science*, pages 120–134. Springer Berlin Heidelberg, 1996.

6 M. J. Gabbay. Foundations of nominal techniques: Logic and semantics of variables in abstract syntax. *Bulletin of Symbolic Logic*, 17(2):161–229, 2011.

7 M. J Gabbay and M. Hofmann. Nominal renaming sets. In I. Cervesato, H. Veith, and A. Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning, 15th International Conference, LPAR 2008, Doha, Qatar, November 22–27, 2008. Proceedings*, volume 5330 of *Lecture Notes in Computer Science*, pages 158–173. Springer, 2008.

8 A. Grothendieck. Pursuing stacks. Manuscript, 1983.

9 M. Hofmann. Syntax and semantics of dependent types. In A. M. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*, Publications of the Newton Institute, pages 79–130. Cambridge University Press, 1997.

10 M. Hofmann and T. Streicher. Lifting Grothendieck universes. Unpublished note, 1999.

11 P T. Johnstone. *Topos Theory*. Number 10 in LMS Mathematical Monographs. Academic Press, London, 1977.

12 C. Kapulkin, P. L. Lumsdaine, and V. Voedodsky. The simplicial model of univalent foundations. *ArXiv e-prints*, arXiv:1211.2851 [math.LO], November 2012.

13 P. L. Lumsdaine and M. A. Warren. The local universes model: an overlooked coherence construction for dependent type theories. *ArXiv e-prints*, arXiv:1411.1736 [math.LO], November 2014.

14 A. M. Pitts. An equivalent presentation of the Bezem-Coquand-Huber category of cubical sets. *ArXiv e-prints*, arXiv:1401.7807, December 2013.

15 A. M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*, volume 57 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press, 2013.

16 R. A. G. Seely. Locally cartesian closed categories and type theories. *Math. Proc. Cambridge Philos. Soc.*, 95:33–48, 1984.

17 S. Staton. *Name-Passing Process Calculi: Operational Models and Structural Operational Semantics*. PhD thesis, University of Cambridge, 2007. Available as University of Cambridge Computer Laboratory Technical Report Number UCAM-CL-TR-688.

18 S. Staton. Completeness for algebraic theories of local state. In L. Ong, editor, *Foundations of Software Science and Computational Structures*, volume 6014 of *Lecture Notes in Computer Science*, pages 48–63. Springer Berlin Heidelberg, 2010.

19 T. Streicher. Universes in toposes. In L. Crosilla and P. Schuster, editors, *From Sets and Types to Topology and Analysis, Towards Practicable Foundations for Constructive Mathematics*, volume 48 of *Oxford Logic Guides*, chapter 4, pages 78–90. Oxford University Press, 2005.

20 A. Swan. An algebraic weak factorisation system on 01-substitution sets: A constructive proof. *ArXiv e-prints*, arXiv:1409.1829, September 2014.

21 The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations for Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

On Extensionality of $\lambda*$

Andrew Polonsky

Laboratoire PPS, Université Paris-Diderot – Paris 7, Paris, France
andrew.polonsky@gmail.com

Abstract

We prove an extensionality theorem for the "type-in-type" dependent type theory with Σ-types. We suggest that in type theory the notion of extensional equality be identified with the logical equivalence relation defined by induction on type structure.

1998 ACM Subject Classification F.4.1 Mathematical Logic

Keywords and phrases Extensionality, logical relations, type theory, lambda calculus, reflection

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.221

1 Introduction

Our goal is to find a formulation of type theory with extensional equality in such a way that the essential features of type theory – including canonicity, normalization, and decidability of type checking – remain valid.

This paper is the first in a series where we will pursue this goal using a syntactic methodology. Specifically, we will define the extensional equality type by induction on type structure, so that, for example, the equality on Π- and Σ-types is definitionally equal to

$$f \simeq_{\Pi x:A.B(x)} f' \quad = \quad \prod_{xx':A} \prod_{x^*:x\simeq_A x'} fx \simeq_{B(x^*)} f'x' \tag{1}$$

$$(a,b) \simeq_{\Sigma x:A.B(x)} (a',b') \quad = \quad \sum_{x^*:a\simeq_A a'} b \simeq_{B(x^*)} b' \tag{2}$$

The view that the notion of extensional equality in type theory should be identified with the equivalence relation so defined has extensive presence in the literature, going back at least to Tait [9] and Altenkirch [1].

This position is to be contrasted with another prominent view, which takes extensional equality to mean the adjunction to Martin-Löf intensional identity type $Id_A(x,y)$ of the propositional reflection rule

$$\frac{p : Id_A(s,t)}{s = t : A}$$

This inference rule commits treason against some fundamental design principles of type theory, resulting in the failure of normalization and related pathologies.

We therefore begin with a moment of "full disclosure":

Extensionality Thesis. The extensional equality of type theory is the logical equivalence relation between elements of the term model defined by induction on type structure.

Before we lay out in greater detail our program to convert this philosophical thesis into a mathematical definition, we invite the reader to consider what the final result may be expected to look like.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 221–250

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

First and foremost – we seek a canonical way to associate to every type A, and any two terms of type A, a (possibly new) type of *extensional equalities* between the two terms. In other words, given expressions A, a, a', we wish to define, or construct, a new expression $a \simeq_A a'$ validating the following rule:

$$\frac{A : *\qquad a : A\qquad a' : A}{a \simeq_A a' : *}$$

(In this paper, $*$ denotes the universe of types.)

So far, we have redisplayed the formation rule for the intensional identity type with a new symbol. The difference here is that, in general, *we do not require $a \simeq_A a'$ to be a new type constructor.* We expect that the $\simeq_A$-symbol may well behave like a defined function, so that it *reduces* to previously defined types, according to the top symbol of A. This is the situation encountered in (1) and (2).

At the same time, we want to be able to iterate the $\simeq$-operation, so that we can form the types $a \simeq_A a'$, $p \simeq_{a \simeq_A a'} p'$, $\alpha \simeq_{p \simeq_{a \simeq_A a'} p'} \alpha'$, etc.

What properties should such a "globular type family" satisfy?

Intuitively, the key properties of equality are:

1. Equality is preserved by every construction of the language.

2. Equality forms a (higher-dimensional) equivalence relation.

The first property, congruence, may be schematically rendered as:

$$\frac{\Gamma, x : A \vdash t(x) : B\qquad \Gamma \vdash a^* : a \simeq_A a'}{\Gamma \vdash \text{“}t(a^*)\text{”} : t(a) \simeq_B t(a')}$$

Note that in presence of type dependency, even the statement of the above rule becomes hardly trivial, since the type $B = B(x)$ of $t(x)$ might itself depend on x.

The second property states that $\simeq_A$ is reflexive, symmetric, transitive on every "level", and that adjacent levels interact correctly. Altogether, these data may be captured neatly by the *Kan filling condition* from homotopy theory, stating that the above globular structure of equalities forms a weak ω-groupoid.

All of the needed properties above are encapsulated in five axioms isolated by Coquand [5]. Accordingly, the type $a \simeq_A a'$ must admit the following operations:

$$\begin{array}{rrcl}
(a : A) & \mathsf{r}(a) & : & a \simeq_A a \\
(x : A \vdash B(x) : \mathbf{Type}) & \mathtt{transp} & : & B(a) \to (a \simeq_A a') \to B(a') \\
(b : B(a)) & \mathtt{Jcomp} & : & \mathtt{transp}\, b\, \mathsf{r}(a) \simeq_{B(a)} b \\
(a : A) & \pi_a & : & \mathtt{isContr}(\Sigma x{:}A.a \simeq_A x) \\
(A, B(x), f, g) & \mathtt{FE} & : & (\Pi x{:}A)\ fx \simeq_{B(x)} gx \ \to\ f \simeq_{\Pi x:A.B(x)} g
\end{array}$$

And now we have obtained a well-defined mathematical problem:
To define an extension of type theory with a new type $a \simeq_A a'$, so that all of Coquand's axioms are satisfied, and the usual metatheoretic properties of type theory remain valid.

Equality from logical relations

The point of departure of all syntactic approaches to extensional equality is to define an equivalence relation on the term model of type theory by induction, where the relation associated to each type constructor is given inductively by the logical condition for that type.

The *logical relation principle* then allows one to derive that every term of type theory preserves this relation, so that the relation is indeed a congruence.

As a simple example, let us consider the universe of simple types.

$$\begin{aligned}&\mathsf{Inductive}\ \mathcal{U} : * \ :=\\ &\quad \mid \circledcirc : \mathcal{U}\\ &\quad \mid \ominus : \mathcal{U} \to \mathcal{U} \to \mathcal{U}\\ &\quad \mid \otimes : \mathcal{U} \to \mathcal{U} \to \mathcal{U}\end{aligned}$$

To each element $A : \mathcal{U}$, we can associate the type of *terms of type* A, by defining a $*$-valued *decoding function* $\mathcal{T} : \mathcal{U} \to *$ by recursion over $\mathcal{U}$:

$$\begin{aligned}\mathcal{T}(\circledcirc) &= N\\ \mathcal{T}(\ominus AB) &= \mathcal{T}A \to \mathcal{T}B\\ \mathcal{T}(\otimes AB) &= \mathcal{T}A \times \mathcal{T}B\end{aligned}$$

We think of $\mathcal{T}A$ as giving the interpretation of object types $A : \mathcal{U}$ on the metalevel.

In just the same way, we may define a family of relations $\{R_A : \mathcal{T}A \to \mathcal{T}A \to *\}$ by recursion over $A : \mathcal{U}$:

$$\begin{aligned}R_{\circledcirc} mn &= Id_N(m,n)\\ R_{\ominus AB} ff' &= \Pi x{:}\mathcal{T}A \Pi x'{:}\mathcal{T}A.\ R_A xx' \to R_B(fx)(f'x')\\ R_{\otimes AB} pp' &= R_A(\pi_1 p)(\pi_1 p') \times R_B(\pi_2 p)(\pi_2 p')\end{aligned}$$

By induction on A, we can verify that R_A is indeed an equivalence relation. Moreover, in contrast to the Id_A-relation, R_A actually validates the rule of function extensionality – pretty much just by definition!

Thus we are confronted with a possibility that the intensional character of type theory may be quashed simply by declaring the logical equivalence relation to be the "right" notion of equality, and the Id-type to be no good, ugly, bad.

All that remains is to extend the previous construction to dependent type theory, and make extensional equality part of our language.

Alas, while the logical relations argument can indeed be extended to the dependent setting – and we will treat this procedure in some detail – the additional requirement is oxymoronic as stated.

Logical relations are an external concept – they are valued in types of the meta-level. Their very construction requires the language to be a "closed" object, since it takes place in a metatheory where we have an interpretation of the given system $(\mathcal{U}, \mathcal{T})$ of types.

One might imagine some kind of a reflection procedure, whereby the values of the relation are reincarnated as elements of $\mathcal{U}$ again.

But what should that mean, exactly? If we ever add new types into $\mathcal{U}$, we change the domain of the relation being reflected. Is there a universe $\mathcal{U}$ which is "closed under" its own logical relation?

Among our contributions here is to clarify what indeed it could all mean, how one might go about constructing such a $\mathcal{U}$, and what one can do with the result. Toward the end of the paper, we produce a candidate universe containing the extensional equality relation for all *closed* types. Verification of the logical relation principle (congruence) for this universe will be treated in subsequent work.

Taking the long view, the promise of a syntactic foundation of extensional equality will be fulfilled whenever the following objectives are attained.

1. Find a type theory in which the universe $\mathcal{U}$ of types reflects the canonical logical relation defined by induction on type structure.
2. Prove extensionality theorem, that every term preserves this relation. Conclude that every ground term is equal to itself (reflexivity). Obtain extensional equality as the relation induced by the reflexive equality of a type with itself.
3. Extend the reflection mechanism to open terms.
4. Add operators to witness the Kan filling condition.
5. Treat the remaining axioms of Coquand.

Contents of the paper

Here we will deliver on the first goal in the above sequence. Starting from the simply typed lambda calculus, we proceed in a systematic way to extend our theory until we arrive at a universe which already contains the extensional relation inside its type hierarchy.

The paper can thus be read as a kind of "bootstrapping procedure" for the plan above. We show from first principles, how to define, by following a set pattern, a minimal type system containing the extensional equality type for all closed types. Further properties of extensional equality, including extension to open terms, can be built up using this system as a base.

An intermediate stage in our development is the treatment of logical relations for dependent types. For this purpose, we choose to work with the inconsistent pure type system $\lambda*$. In our view, this is by far and away the simplest formulation of dependent type theory, and tuning out the "universe management noise" allows us to see more clearly the computational relationships between the types and extensional equalities. [1]

After finding the candidate universe, we apply the usual method of stratification to remove the inconsistency inherited from $\lambda*$. We conjecture that the stratified system is strongly normalizing.

One can also read this paper without regard to the program above, as presenting a generalization of the extensionality theorem of Tait [9] to the dependent setting.

While logical relations for dependent types have been treated by various authors ([2], [4], [8]), our presentation contains some novel features. Specifically, we use a generalization of Dybjer's indexed inductive-recursive definitions to encapsulate the data pertaining to (heterogeneous) dependent relations associated to equalities between types. Our treatment of universes also differs from e.g. parametricity theory in that we instantiate the relation on the universe with the least congruence between types.

In particular, in our approach, every relation between types is necessarily an equivalence in the sense of homotopy type theory.

2 The simply typed case

We begin by stating the extensionality theorem for the simply typed λ-calculus.

The syntax of simple types and typed terms is as follows:

$$\mathbb{T} = o \mid \mathbb{T} \to \mathbb{T} \mid \mathbb{T} \times \mathbb{T}$$
$$\Lambda = x \mid \lambda x{:}\mathbb{T}.\Lambda \mid \Lambda\Lambda \mid (\Lambda, \Lambda) \mid \pi_1\Lambda \mid \pi_2\Lambda$$

[1] It was also attractive to work in a system where types and terms are treated completely homogeneously, if only as a "sanity check" that we are not making ad hoc definitions based on the "kind" of the object.

A model of $\lambda_{\to}$ consists of a family of sets $\{X_A \mid A \in \mathbb{T}\}$ where

$$X_{A\to B} \subseteq X_A^{X_B}$$
$$X_{A\times B} \subseteq X_A \times X_B$$

are such that $X_{A\to B}$ is closed under abstraction of terms of type B over variables of type A, and $X_{A\times B}$ is closed under pairs of definable elements of X_A and X_B.

The interpretation of types is given by

$$[\![A]\!] = X_A$$

The interpretation of terms is parametrized by an environment $\rho = \{\rho_A : V_A \to X_A\}$, assigning elements of the domain to the free variables of the term.

Let Env be the set of such collections of functions.

A term $t : A$ is interpreted as a map $[\![t]\!] : \mathsf{Env} \to [\![A]\!]$. We write $[\![t]\!]_\rho$ for $[\![t]\!](\rho)$. The definition of $[\![t]\!]_\rho$ is given by induction:

$$\begin{aligned}
[\![x : A]\!]_\rho &= \rho_A(x)\\
[\![st]\!]_\rho &= [\![s]\!]_\rho [\![t]\!]_\rho\\
[\![\lambda x{:}A.t]\!]_\rho &= (a \mapsto [\![t]\!]_{\rho, x:=a})\\
[\![(s,t)]\!]_\rho &= ([\![s]\!]_\rho, [\![t]\!]_\rho)\\
[\![\pi_i t]\!]_\rho &= a_i, \text{ where } [\![t]\!]_\rho = (a_1, a_2) \in [\![A_1 \times A_2]\!]
\end{aligned}$$

A relation $R = \{R_A \mid R_A \subseteq [\![A]\!] \times [\![A]\!], A \in \mathbb{T}\}$ is said to be *logical* if

$$\begin{aligned}
R_{A\to B} f f' &\iff \forall a a' {:} X_A . R_A a a' \Rightarrow R_B (f a)(f' a')\\
R_{A\times B}(a,b)(a',b') &\iff R_A a a' \wedge R_B b b'
\end{aligned}$$

Here and throughout, (two-sided) double arrows represent logical implication (equivalence) on the meta-level.

▶ **Theorem 1** (Extensionality Theorem). *Let R be logical. Suppose that t is a typed term:*

$$x_1 : A_1, \ldots, x_n : A_n \vdash t : T$$

and let there be given

$$a_1, a_1' \in [\![A_1]\!], \ldots, a_n, a_n' \in [\![A_n]\!]$$

Then

$$R_{A_1} a_1 a_1', \ldots, R_{A_n} a_n a_n' \implies R_T [\![t]\!]_{\vec{x}:=\vec{a}} [\![t]\!]_{\vec{x}:=\vec{a}'}$$

In other words, every typed λ-term induces a function which maps related elements to related elements. As a corollary, we get that a closed term $t \in \Lambda^0(A)$ is R_A-related to itself.

The proof of the above theorem proceeds by induction on the structure of derivation that $t : T$. For illustration, we treat the abstraction case $t = \lambda x{:}A.t'$. Suppose we have

$$\frac{x_1 : A_1, \ldots, x_n : A_n, x : A \vdash t' : B}{x_1 : A_1, \ldots, x_n : A_n \vdash \lambda x{:}A.t' : A \to B}\ \text{Abs}$$

Let $(a_1,\ldots,a_n),(a'_1,\ldots,a'_n) \in [\![A_1]\!] \times \cdots \times [\![A_n]\!]$ be such that $R_{A_i} a_i a'_i$. Assume $a, a' \in [\![A]\!]$ are given, and suppose that $R_A a a'$.

By induction hypothesis,

$$R_B [\![t']\!]_{x_1\ldots x_n x:=a_1\ldots a_n a} [\![t']\!]_{x_1\ldots x_n x:=a'_1\ldots a'_n a'}$$

which can be rewritten as

$$R_B [\![\lambda x.t']\!]_{x_1\ldots x_n:=a_1\ldots a_n}(a) [\![\lambda x.t']\!]_{x_1\ldots x_n:=a'_1\ldots a'_n}(a')$$

Since a, a' were arbitrary, and $R_{A\to B}$ is logical, it follows that

$$R_{A\to B} [\![\lambda x.t']\!]_{x_1\ldots x_n:=a_1\ldots a_n} [\![\lambda x.t']\!]_{x_1\ldots x_n:=a'_1\ldots a'_n}$$

The other cases are treated similarly.

We remark that the structure of the proof that $R_T tt$ recapitulates rather precisely the structure of t itself. In particular, the theorem is completely constructive. Anticipating upcoming development, consider a constructive reading of the theorem's statement:

From a proof a_1^* that $R_{A_1} a_1 a'_1$
and a proof a_2^* that $R_{A_2} a_2 a'_2$
...
and a proof a_n^* that $R_{A_n} a_n a'_n$
Get a proof $t(a_1^*,\ldots,a_n^*)$
that $R_T t(a_1,\ldots,a_n) t(a'_1,\ldots,a'_n)$

This motivates us to think of the above extensionality property as an operation which, given terms which relate elements in the context, substitutes these connections into t to get a relation between the corresponding instances of t.

In this interpretation, the proof that a closed term t is related to itself

$$\mathsf{r}(t) : R_T tt$$

has specific computational content. Furthermore, the algorithm associated to this proof has the same structure as t itself.

Given a relation R_o on X_o, we can extend it to the full structure by *defining* $R_{A\to B}$, $R_{A\times B}$ to be such as to satisfy the logical conditions; the resulting family then satisfies the theorem by construction.

3 The dependent case

To make matters simple, we work with the pure type system (PTS) formulation of dependent type theory with "type-in-type" extended by Σ-types. This system is denoted as λ∗. It has a universal type $*$, the type of all types. This allows us to unify into one the three classical judgement forms of dependent type theory:

$\Gamma \vdash A$ **Type**

$\Gamma \vdash a : A$

$\Gamma \vdash B : (A)\mathbf{Type}$

The judgment $\Gamma \vdash A$ **Type** is replaced by $\Gamma \vdash A : *$. Similarly, $\Gamma \vdash (A)B$ **Type** is replaced by $\Gamma, x{:}A \vdash B : *$. Thus types and terms of type $*$ are completely identified.

The syntax of λ∗ is given in Figure 1.

$$A, t \quad ::= \quad * \mid x \mid \Pi x{:}A.B \mid \Sigma x{:}A.B \mid \lambda x{:}A.t \mid st \mid (s,t) \mid \pi_1 t \mid \pi_2 t$$

$$\frac{}{\vdash * : *}$$

$$\frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : *}{\Gamma, y : B \vdash M : A}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : *}{\begin{array}{c}\Gamma \vdash \Pi x{:}A.B : * \\ \Gamma \vdash \Sigma x{:}A.B : *\end{array}}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : * \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B}$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash f : \Pi x{:}A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[a/x]}$$

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a,b) : \Sigma x{:}A.B}$$

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash p : \Sigma x{:}A.B}{\begin{array}{c}\Gamma \vdash \pi_1 p : A \\ \Gamma \vdash \pi_2 p : B[\pi_1 p/x]\end{array}}$$

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : * \qquad A = B}{\Gamma \vdash M : B}$$

$$\begin{array}{rcl}(\lambda x{:}A.s)t & \longrightarrow & s[t/x] \\ \pi_1(s,t) & \longrightarrow & s \\ \pi_2(s,t) & \longrightarrow & t\end{array}$$

Figure 1 The system $\lambda *$.

Our choice of rules differs from the standard definition of $\lambda *$ as a PTS in that the introduction and elimination rules (for both Π- and Σ-types) come with information on the well-formedness of the type arguments A, B. This however does not change the set of derivable sequents.
The proof of this essentially appears already in Barendregt [3], see Corollary 5.2.14(1,2) and Lemma 5.2.13(3). The argument there is equally applicable to Σ as to Π. (And in $\lambda *$, it becomes even easier.)

Notation. The parentheses following the matrix of the Π, Σ, and λ constructors are not part of the syntax, and merely pronounce the fact that the term may depend on the variables in question. In general, when we write $t = t(x_1, \ldots, x_n)$, we do not commit to having displayed all the free variables of t; it is never mandatory to display a free variable.

The purpose of this notation is merely to reduce clutter in anticipation of substitution of t by an instance of (some of the) variables. Our general notation for capture-avoiding substitution of a free variable x in t by a term a is

$$t[a/x]$$

In particular, if $t = t(x_1, \ldots, x_n)$, then

$$t[a_1/x_1]\cdots[a_n/x_n] = t(a_1, \ldots, a_n)$$

In the following development, we shall consider the open term model of the above type theory, using the same theory as our meta-level. To simplify notation, we write $[\![A]\!]$ simply as A. As well, if $t(x_1, \ldots, x_n) : T(x_1, \ldots, x_n)$, then $[\![t]\!]_{x_1,\ldots,x_n:=a_1,\ldots,a_n}$ is denoted as $t(a_1, \ldots, a_n)$.

From now on, we work in λ∗.

The only axiom of this type system has the form $* : *$, asserting that the universe of types $*$ is itself a type. Its intuitive meaning is

> The collection of structures, which types are interpreted by, forms the same kind of structure.

In particular, if types are interpreted by types-with-relation $R_A : A \to A \to *$, then this interpretation must also include a relation on the universe of types

$$R_* : * \to * \to *$$

But how should this relation interact with objects inhabiting related types? Is there a logical condition for the universe? What is a dependent logical relation?

To answer these questions, we engage in a series of thought experiments.

First, we stipulate that, dependent logical relation, whatever it finally turns out to be, must at the very least be such that the congruence (extensionality) rule is always valid.

Let us then consider how the previous extensionality theorem could be extended to the dependent case. Suppose we have terms

$$x{:}A \vdash B(x) : * \tag{3}$$

$$x{:}A \vdash b(x) : B(x) \tag{4}$$

If we are now given $a : A$, $a' : A$, we want to conclude that

$$R_A a a' \to R_B b(a) b(a') \tag{5}$$

However, the two terms $b(a)$ and $b(a')$ have different types! Evidently, we need more structure to formulate extensionality of dependent maps. But where should this structure come from?

Looking again at (3), let us first consider extensionality of the term $B(x)$:
From any witness a^* of the hypothesis of (5), it must be possible to construct a witness $B(a^*)$ to the relation $R_* B(a) B(a')$.

Thinking of $B(a^*)$ as encoding some kind of correspondence between types, we can imagine the relation $b(a^*)$ between $b(a)$ and $b(a')$ to be "lying over" the relation $B(a^*)$ between $B(a)$ and $B(a')$.

This suggests the following principle:

Every witness $e : R_* AB$ to the fact that A and B are related elements of the universe induces a relation

$$\sim e : A \to B \to *$$

between elements of corresponding types.

Notation. Put

$$\begin{array}{rcll} a \simeq_A a' & := & R_A a a' & a, a' : A \\ A \simeq B & := & R_* AB & A, B : * \\ a \sim_e b & := & \sim eab & a{:}A, b{:}B, e : A \simeq B \end{array}$$

We call $e : A \simeq B$ a *type equality* between A and B, and $a \sim_e b$ a *dependent equality*, or *heterogeneous equality induced by* e.

Notice that $(A \simeq B) = (A \simeq_* B)$.

The considerations thus far have yielded that

$$\frac{x{:}A \vdash B(x) : * \qquad \vdash a^* : a \simeq_A a'}{B(a^*) : B(a) \simeq B(a')} \qquad \frac{x{:}A \vdash b(x) : B(x) \qquad \vdash a^* : a \simeq_A a'}{b(a^*) : b(a) \sim_{B(a^*)} b(a')}$$

Now suppose $x \notin B(x) = B$. Then, for $a^* : a \simeq_A a'$, we have

$$B() = B(a^*) : B(a) \simeq B(a')$$

$$B() : B \simeq B$$

So $B(a^*)$, for $x \notin B(x)$, gives us a type equality of B with itself. We call such equality the *identity on* B; it corresponds to the identity equivalence in homotopy type theory.

The relation $\sim_{B()} : B \to B \to *$ induced by the identity equivalence we call the *extensional equality on type* B, and we define

$$a \simeq_A a' \quad := \quad a \sim_{A()} a'$$

In particular, since $x \notin * : *$, we have

$$*() : * \simeq *$$

$$*() : * \sim_{*()} *$$

$$A \simeq_* B \quad = \quad A \sim_{*()} B \quad = \quad A \simeq A$$

so that extensional equality on the universe is type equality and is the relation induced by the identity equivalence of the universe with itself.

With these definitions, we can make sense of the "logical conditions"

$$R_{\Pi x:A.B(x)} f f' = \prod_{a:A} \prod_{a':A} \Pi a^* : R_A a a'.\, (fa) \sim_{B(a^*)} (f' a') \tag{6}$$

$$R_{\Sigma x:A.B(x)} p p' = \Sigma a^* : R_A\, (\pi_1 p)\, (\pi_1 p').\, (\pi_2 p) \sim_{B(a^*)} (\pi_2 p') \tag{7}$$

However, as the considerations illustrate, we may also just dispense with the relation $(R_A) = (\simeq_A)$ altogether and assume as primitive only the following two objects:

- A binary relation on the universe:

$$\frac{A : * \qquad B : *}{A \simeq B : *}$$

- For all types A, B, and for every proof $e : A \simeq B$ that they are related, a binary relation between A and B:

$$\frac{e : A \simeq B \qquad a : A \qquad b : B}{a \sim_e b : *}$$

The question of what it means for a pair $(\simeq, \sim_e)$ to be logical will be answered in Section 5. It will allow us to prove the generalization of extensionality theorem of the following shape:

▶ **Theorem 2.** *Let $(\simeq, \sim_e)$ be logical. For every term t typed in the context*

$$x_1 : A_1, \cdots, x_n : A_n(x_1, \ldots, x_{n-1}) \vdash t(x_1, \ldots, x_n) : T(x_1, \ldots, x_n)$$

and for any sequence of coordinate-wise related instances

$$a_1 : A_1, \cdots, a_n : A_n(a_1, \ldots, a_{n-1})$$
$$a'_1 : A_1, \cdots, a'_n : A_n(a'_1, \ldots, a'_{n-1})$$
$$a^*_1 : a_1 \sim_{A_1()} a'_1, \cdots, a^*_n : a_n \sim_{A_n(a^*_1, \ldots, a^*_{n-1})} a'_n$$

there is a witness $t(a^*_1, \ldots, a^*_n)$ to the fact that

$$t(a_1, \ldots, a_n) \sim_{T(a^*_1, \ldots, a^*_n)} t(a'_1, \ldots, a'_n).$$

As it appears now, the statement is perhaps unparsable, since the conclusion already makes reference to the result of substituting $a^*_1, \ldots, a^*_n$ into $T(\vec{x})$, which requires the extensionality of the judgement $\Gamma \vdash T(\vec{x}) : *$ to be known beforehand. The proof of this latter fact will again depend on extensionality of subterms appearing in T.

Also, the definitions of $\sim$ and $\simeq$ themselves, will evidently depend on being able to perform the "cell substitution" given by the theorem (as in the case of the equality for Π- and Σ-types).

In order to cut through all this circularity, and to formulate all the necessary concepts precisely, we first move to represent the type universe of λ* in a minimal extension of the system relevant for this purpose. The above theorem will be stated for the result of reflecting the meta-level into this universe. The next step is to mutually define the type of equivalences between two elements of this universe, and the corresponding relations induced by such equivalences. The inter-dependency between these concepts is resolved using an indexed inductive–recursive definition of Dybjer and Setzer [6], and this allows us to state the above theorem for the (reflected) universe. Finally, we prove the theorem by induction on the structure of derivations.

4 λ* in λ*

To motivate the inductive–recursive definition of the universe of λ*, let us recall the representation of simply typed lambda calculus.

Inductive $\mathcal{U} : *$:=
$\quad | \circledcirc : \mathcal{U}$
$\quad | \ominus : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$
$\quad | \otimes : \mathcal{U} \to \mathcal{U} \to \mathcal{U}$

$$\begin{aligned}\mathcal{T} &: \mathcal{U} \to *\\ \mathcal{T}(\circledcirc) &= A_0\\ \mathcal{T}(\ominus AB) &= \mathcal{T}A \to \mathcal{T}B\\ \mathcal{T}(\otimes AB) &= \mathcal{T}A \times \mathcal{T}B\end{aligned}$$

It is straightforward to define, for every simple type $A \in \mathbb{T}$, a corresponding term $\overline{A}$ of type $\mathcal{U}$. Furthermore, for every valid $\lambda_{\to\times}$ judgment $\Gamma \vdash M : A$, there is a term $\overline{M}$ of type $\mathcal{T}\overline{A}$ and a derivation of $\overline{\Gamma} \vdash \overline{M} : \mathcal{T}\overline{A}$, where contexts are translated as

$$\overline{x_1 : A_1, \cdots, x_n : A_n} = x_1 : \mathcal{T}\overline{A_1}, \cdots, x_n : \mathcal{T}\overline{A_n}$$

Next, consider the family R_A, for $A : \mathcal{U}$, defined on page 223. Clearly, this family gives a logical relation on the interpretation of $\lambda_{\to\times}$ in $(\mathcal{U}, \mathcal{T})$. Thus we may verify the following

▶ **Proposition 3.** *Suppose* $x_1 : A_1, \ldots, x_n : A_n \vdash_{\lambda_{\to\times}} M : A$.
Consider the context consisting of variables $x_1, x_1' : T\overline{A_1}, \ldots, x_n, x_n' : T\overline{A_n}$ *and*

$$x_1^* : R_{A_1} x_1 x_1', \ldots, x_n^* : R_{A_n} x_n x_n',$$

There is a term $M^* = M^*(x_1, x_1', x_1^*, \ldots, x_n, x_n', x_n^*)$ *such that*

$$\{x_1, x_1', x_1^*, \ldots, x_n, x_n', x_n^*\} \vdash M^* : R_A\, \overline{M}\, \overline{M'}$$

where $M' = M[x_1'/x_1, \ldots, x_n'/x_n]$.

(The proof proceeds exactly as before, replacing set-theoretic concepts by their type-theoretic counterparts.)

In trying to repeat the above proposition for dependent type theory, we run into a problem already when trying to represent the universe of types. As becomes evident, the collection of types must be defined *simultaneously* with the decoding function:

$$\begin{aligned}&\mathsf{Inductive}\ \mathcal{U} : * \ :=\\ &\qquad \mid \circledcirc : \mathcal{U}\\ &\qquad \mid \textcircled{\Pi} : \Pi A : \mathcal{U}.(\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\ &\qquad \mid \textcircled{\Sigma} : \Pi A : \mathcal{U}.(\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\ &\quad \mathsf{with}\ \ \mathcal{T} : \mathcal{U} \to * \ :=\\ &\qquad \mathcal{T}(\circledcirc) = A_0\\ &\qquad \mathcal{T}(\textcircled{\Pi} AB) = \Pi a : \mathcal{T}A.\mathcal{T}[Ba]\\ &\qquad \mathcal{T}(\textcircled{\Sigma} AB) = \Sigma a : \mathcal{T}A.\mathcal{T}[Ba]\end{aligned}$$

The full theory of such inductive-recursive definitions, together with a model construction, is treated in detail by Dybjer and Setzer [6]. Ghani et al. [7] give a modern presentation, considerably generalizing the concept.

Reflection of λ∗

The inductive-recursive definition of the universe $\mathcal{U}$ of $\lambda*$-types is as follows:

$$
\begin{aligned}
&\mathsf{Inductive}\ \mathcal{U} : * \ :=\\
&\qquad\quad \mid \textcircled{Π} : \Pi A : \mathcal{U}.(\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\
&\qquad\quad \mid \textcircled{Σ} : \Pi A : \mathcal{U}.(\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\
&\qquad\quad \mid \circledast : \mathcal{U}\\
&\quad \mathsf{with}\quad \mathcal{T} : \mathcal{U} \to * \ :=\\
&\qquad\quad \mathcal{T}(\textcircled{Π} AB) = \Pi a : \mathcal{T}A.\mathcal{T}[Ba]\\
&\qquad\quad \mathcal{T}(\textcircled{Σ} AB) = \Sigma a : \mathcal{T}A.\mathcal{T}[Ba]\\
&\qquad\quad \mathcal{T}(\circledast) = \mathcal{U}
\end{aligned}
$$

Let $\lambda*\mathcal{U}$ be $\lambda*$ augmented with the above datatype. Notice that every derivation in $\lambda*$ is also a derivation in $\lambda*\mathcal{U}$.

▶ **Definition 4.** We define a map $\overline{(\cdot)}$ from the raw terms of $\lambda*$ to the raw terms of $\lambda*\mathcal{U}$ as follows:

$$
\begin{aligned}
\overline{*} &= \circledast\\
\overline{x} &= x\\
\overline{\Pi x{:}A.B} &= \textcircled{Π}\overline{A}(\lambda x{:}\mathcal{T}\overline{A}.\overline{B})\\
\overline{\Sigma x{:}A.B} &= \textcircled{Σ}\overline{A}(\lambda x{:}\mathcal{T}\overline{A}.\overline{B})\\
\overline{\lambda x{:}A.t} &= \lambda x{:}\overline{A}.\overline{t}\\
\overline{st} &= \overline{s}\overline{t}\\
\overline{(s,t)} &= (\overline{s},\overline{t})\\
\overline{\pi_i s} &= \pi_i \overline{s}
\end{aligned}
$$

▶ **Definition 5.** Let $\Gamma = \{x_1 : A_1, \ldots, x_n : A_n\}$ be a context in $\lambda*$. We define[2]

$$\overline{\Gamma} := \{x_1 : \mathcal{T}\overline{A}_1, \ldots, x_n : \mathcal{T}\overline{A_n}\}$$

▶ **Lemma 6** (Substitution Lemma)**.** *Let M, N be $\lambda*$-terms. Then*

$$\overline{M[N/x]} = \overline{M}[\overline{N}/x]$$

Proof. This is manifest from the fact that $\overline{\cdot}$ is defined completely compositionally. ◀

▶ **Corollary 7.** *Let M, N be $\lambda*$-terms. Then*

$$M = N \quad \Longrightarrow \quad \overline{M} = \overline{N}$$

---

[2] Note that we cannot yet conclude that this definition yields a valid context: so far the $\overline{A}_i$ are just raw terms, and we have not checked that

$$x_1 : \mathcal{T}\overline{A_1}, \ldots, x_i : \mathcal{T}\overline{A_i} \vdash \overline{A_{i+1}} : \mathcal{U}$$

for $0 \le i < n$. As a matter of fact, this will follow from the theorem we are about to prove, but for now we just treat $\overline{\Gamma}$ as a "raw context".
(Indeed, the typing rules of $\lambda*$, like all PTSs, do not include context hygiene, because it is enforced implicitly via the hypotheses of context-extending rules.)

▶ **Theorem 8** (Reflection of $*$ into $\mathcal{U}$).

$$\Gamma \vdash_{\lambda *} M : A \implies \overline{\Gamma} \vdash_{\lambda *\mathcal{U}} \overline{M} : \mathcal{T}\overline{A}$$

Proof. The translation is done by induction on $\Gamma \vdash M : A$.
Axiom. $\vdash_{\lambda *} * : *$. Then $\overline{\Gamma} = \Gamma = \langle\rangle$. Also $\overline{A} = \overline{M} = \circledast$. The conversion rule gives

$$\frac{\circledast : \mathcal{U} \qquad \mathcal{T}\circledast : * \qquad \mathcal{U} = \mathcal{T}\circledast}{\circledast : \mathcal{T}\circledast}$$

Thus indeed $\vdash_{\mathcal{U}} \overline{M} : \mathcal{T}\overline{A}$.
Variable. Suppose the derivation ends with

$$\frac{\Gamma \vdash A : *}{\Gamma, x : A \vdash x : A}$$

By induction hypothesis, we have

$$\overline{\Gamma} \vdash \overline{A} : \mathcal{T}\overline{*}$$

Since $\overline{*} = \circledast$, we have $\Gamma \vdash \overline{A} : \mathcal{T}\circledast$. Then $\overline{\Gamma} \vdash \overline{A} : \mathcal{U}$, and $\overline{\Gamma} \vdash \mathcal{T}\overline{A} : *$.
By the variable rule, we have

$$\Gamma, x : \mathcal{T}\overline{A} \vdash x : \mathcal{T}\overline{A}$$

Weakening. Let the derivation end with

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : *}{\Gamma, y : B \vdash M : A}$$

By induction hypothesis, we have

$$\overline{\Gamma} \vdash \overline{M} : \mathcal{T}\overline{A}$$
$$\overline{\Gamma} \vdash \overline{B} : \mathcal{T}\circledast$$

That is, $\overline{\Gamma}$ yields $\vdash \overline{B} : \mathcal{U}$. Then $\mathcal{T}\overline{B} : *$. By weakening,

$$\overline{\Gamma}, y : \mathcal{T}\overline{B} \vdash \overline{M} : \mathcal{T}\overline{A}$$

Π-formation. Given

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : *}{\Gamma \vdash \Pi x{:}A.B : *}$$

the induction hypotheses yield

$$\overline{\Gamma} \vdash \overline{A} : \mathcal{T}\overline{*} \tag{8}$$
$$\overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \overline{B} : \mathcal{T}\overline{*} \tag{9}$$

Since $\overline{A}, \overline{B} : \mathcal{T}\overline{*} = \mathcal{T}\circledast = \mathcal{U}$, we have $\overline{\Gamma} \vdash \mathcal{T}\overline{A} : *$ as well as $\Gamma, x : \mathcal{T}\overline{A} \vdash \mathcal{T}\overline{B} : *$.
By the Π-introduction rule, (9) yields

$$\overline{\Gamma} \vdash \lambda x{:}\mathcal{T}\overline{A}.\overline{B} : \mathcal{T}\overline{A} \to \mathcal{U}$$

whence Π-elimination together with (8) yields

$$\overline{\Gamma} \vdash \textcircled{\Pi}\overline{A}(\lambda x{:}\mathcal{T}\overline{A}.\overline{B}) : \mathcal{U}$$

That is,

$$\overline{\Gamma} \vdash \overline{\Pi x{:}A.B} : \mathcal{T}\overline{*}$$

Σ-formation. Treated in an analogous fashion.

Π-introduction. Suppose the derivation is of the form

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : * \qquad \Gamma, x : A \vdash b : B}{\Gamma \vdash \lambda x{:}A.b : \Pi x{:}A.B}$$

The induction hypotheses give us

$$\begin{aligned} \overline{\Gamma} &\vdash \overline{A} : \mathcal{T}\overline{*} \\ \overline{\Gamma}, x : \mathcal{T}\overline{A} &\vdash \overline{B} : \mathcal{T}\overline{*} \\ \overline{\Gamma}, x : \mathcal{T}\overline{A} &\vdash \overline{b} : \mathcal{T}\overline{B} \end{aligned} \tag{10}$$

As in the previous case, we actually have

$$\frac{\overline{\Gamma} \vdash \overline{A} : \mathcal{U} \qquad \overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \overline{B} : \mathcal{U}}{\overline{\Gamma} \vdash \overline{\Pi x{:}A.B} : \mathcal{U}} \qquad \frac{\overline{\Gamma} \vdash \mathcal{T}\overline{A} : * \qquad \overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \mathcal{T}\overline{B} : *}{\overline{\Gamma} \vdash \mathcal{T}[\overline{\Pi x{:}A.B}] : *}$$

By Π-introduction on (10), we have

$$\overline{\Gamma} \vdash \lambda x{:}\mathcal{T}\overline{A}.\overline{b} : \Pi x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B}$$

But we also find that

$$\Pi x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B} = \mathcal{T}[\textcircled{\scriptsize U}\overline{A}(\lambda x{:}\mathcal{T}\overline{A}.\overline{B})] = \mathcal{T}\overline{\Pi x{:}A.B} \tag{11}$$

and so conclude that

$$\overline{\Gamma} \vdash \overline{\lambda x{:}A.b} : \mathcal{T}\overline{\Pi x{:}A.B}$$

Π-elimination. Suppose we are given

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash f : \Pi x{:}A.B \qquad \Gamma \vdash a : A}{\Gamma \vdash fa : B[a/x]}$$

The induction hypothesis yield, on the one hand, that

$$\frac{\overline{\Gamma} \vdash \overline{A} : \mathcal{U} \qquad \overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \overline{B} : \mathcal{U}}{\overline{\Gamma} \vdash \overline{\Pi x{:}A.B} : \mathcal{U}} \qquad \frac{\overline{\Gamma} \vdash \mathcal{T}\overline{A} : * \qquad \overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \mathcal{T}\overline{B} : *}{\overline{\Gamma} \vdash \mathcal{T}[\overline{\Pi x{:}A.B}] : *}$$

and on the other hand, that

$$\begin{aligned} \overline{\Gamma} &\vdash \overline{f} : \mathcal{T}\overline{\Pi x{:}A.B} \\ \overline{\Gamma} &\vdash \overline{a} : \mathcal{T}\overline{A} \end{aligned}$$

Since $\overline{f}$ by conversion in (11) has type $\Pi x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B}$, we may write

$$\overline{\Gamma} \vdash \overline{f}\overline{a} : \mathcal{T}\overline{B}[\overline{a}/x]$$

By Lemma 6, the type in the above judgment is equal to $\mathcal{T}\overline{B[a/x]}$.

Σ-introduction. When we are at

$$\frac{\Gamma \vdash A : * \quad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash a : A \qquad \Gamma \vdash b : B[a/x]}{\Gamma \vdash (a, b) : \Sigma x{:}A.B}$$

the induction hypotheses give as before that

$$\overline{\Gamma} \vdash \mathcal{T}\overline{A} : *$$
$$\overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \mathcal{T}\overline{B} : *$$

and, in addition, we also have

$$\overline{\Gamma} \vdash \overline{a} : \mathcal{T}\overline{A}$$
$$\overline{\Gamma} \vdash \overline{b} : \mathcal{T}\overline{B[a/x]}$$

Recall that

$$\overline{\Sigma x{:}A.B} = \textcircled{\Sigma}\overline{A}(\lambda x{:}\mathcal{T}\overline{A}.\overline{B})$$
$$\mathcal{T}\overline{\Sigma x{:}A.B} = \Sigma x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B}$$

By Lemma 6, $\mathcal{T}\overline{B[a/x]} = \mathcal{T}\overline{B}[\overline{a}/x]$. Hence $b : \mathcal{T}\overline{B}[\overline{a}/x]$.
By Σ-introduction, we now obtain

$$\overline{\Gamma} \vdash (\overline{a}, \overline{b}) : \Sigma x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B}$$

In other words, $\overline{\Gamma} \vdash \overline{(a, b)} : \mathcal{T}\overline{\Sigma x{:}A.B}$.

Σ-elimination. Let there be derived

$$\frac{\Gamma \vdash A : * \qquad \Gamma, x : A \vdash B : * \qquad \Gamma \vdash p : \Sigma x{:}A.B}{\begin{array}{c}\Gamma \vdash \pi_1 p : A \\ \Gamma \vdash \pi_2 p : B[\pi_1 p/x]\end{array}}$$

Assume we have

$$\overline{\Gamma} \vdash \overline{A} : \mathcal{U}$$
$$\overline{\Gamma}, x : \mathcal{T}\overline{A} \vdash \overline{B} : \mathcal{U}$$
$$\overline{\Gamma} \vdash \overline{p} : \mathcal{T}\overline{\Sigma x{:}A.B}$$

We have just seen that $\mathcal{T}\overline{\Sigma x{:}A.B} = \Sigma x{:}\mathcal{T}\overline{A}.\mathcal{T}\overline{B}$. Thus

$$\pi_1\overline{p} : \mathcal{T}\overline{A}$$

$$\pi_2\overline{p} : [\mathcal{T}\overline{B}][\pi_1\overline{p}/x]$$

The subjects of these judgements can be rewritten as $\overline{\pi_i p}$.
Also

$$[\mathcal{T}\overline{B}][\pi_1\overline{p}/x] = [\mathcal{T}\overline{B}][\overline{\pi_1 p}/x] = \mathcal{T}[\overline{B}[\overline{\pi_1 p}/x]] = \mathcal{T}[\overline{B[\pi_1 p/x]}]$$

Thus we have

$$\overline{\Gamma} \vdash \overline{\pi_1 p} : \mathcal{T}\overline{A}$$
$$\overline{\Gamma} \vdash \overline{\pi_2 p} : \mathcal{T}\overline{B[\pi_1 p/x]}$$

Conversion. Suppose we come across

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : * \qquad A = B}{\Gamma \vdash M : B}$$

By induction hypothesis, we have

$$\overline{\Gamma} \vdash \overline{M} : \overline{\mathcal{T}A}$$
$$\overline{\Gamma} \vdash \overline{B} : \mathcal{U}$$

By Lemma 7, we have

$$\overline{A} = \overline{B}$$

But clearly that implies that

$$\overline{\mathcal{T}A} = \overline{\mathcal{T}B}$$

It is likewise clear that

$$\overline{\Gamma} \vdash \overline{\mathcal{T}B} : *$$

By the conversion rule, we comprehend

$$\overline{\Gamma} \vdash \overline{M} : \overline{\mathcal{T}B}$$

This completes the proof of the theorem. ◀

5 Extensionality of λ∗

We work in $\lambda * \mathcal{U}$.

In order to precisely state extensionality theorem using the reflection of types into terms of type $\mathcal{U}$, we must give answers to the following questions:

- What does it mean for two types $A, B : \mathcal{U}$ to be equal?
- What does it mean for two elements to be related by a type equality?

To answer these questions, we continue our reasoning as in Section 3.

There we arrived at the signature

$$\simeq : \mathcal{U} \to \mathcal{U} \to *, \qquad \sim_e : \mathcal{T}A \to \mathcal{T}B \to * \quad \text{for all } e : A \simeq B$$

that a dependent logical relation should possess. But what should be the logical conditions associated to type constructors?

Well, keeping in mind the interpretation of $\simeq$ as type equality, we should at the very least require that $\simeq$ is a congruence with respect to type structure – in other words, that every type constructor preserves type equality.

For example, if we are given type equalities $A^* : A \simeq A'$, $B^* : B \simeq B'$, we should be able to construct terms

$$\times^* A^* B^* \quad : \quad (A \times B) \simeq (A' \times B') \tag{12}$$
$$\to^* A^* B^* \quad : \quad (A \to B) \simeq (A' \to B') \tag{13}$$

Notice that these inferences are valid regardless of what kind of semantics we have in mind for $\simeq$: isomorphism, weak equivalence, exact/strict equality, or even logical equivalence. But it should be a notion of equality, not an arbitrary relation on the universe.

It is likewise clear how the relations induced by (12) and (13) should be given in terms of those for A^* and B^*:

$$
\begin{aligned}
(a,b) \sim_{\times^* A^* B^*} (a',b') &= (a \sim_{A^*} a') \times (b \sim_{B^*} b') \\
f \sim_{\to^* A^* B^*} f' &= \prod_{x:A} \prod_{x':A'} x \sim_{A^*} x' \to fx \sim_{B^*} f'x'
\end{aligned}
$$

We invite the reader to the fun exercise of generalizing the above relation laws to the dependent case, to obtain the *dependent* logicality conditions for the Π- and Σ-types.

Concerning the type constructor $*$, being a nullary constructor, it must also preserve equality. Since it has no arguments, this simply means that it is equal to itself:

$$*^* : * \simeq *$$

The logical condition for the universe must produce, given $A, B : *$, a new type $A \sim_{*^*} B$. What should this type be?

The answers to these questions are displayed on the bottom of this page, where we define the dependent logical relation

$$(\mathcal{E}q(A,B) : *, \mathcal{R}el_{\{A,B\}}(e : \mathcal{E}q(A,B)) : \mathcal{T}A \to \mathcal{T}B \to *)$$

on the universe $(\mathcal{U}, \mathcal{T})$ of reflected $\lambda*$-types.

The key difficulty introduced by type dependency is that the definitions of $\mathcal{E}q(A,B)$ and $\mathcal{R}el_{\{A,B\}}(e)$ cannot be disentangled from one another: they must be defined simultaneously. This naturally suggests that their interdependency could be captured using a variant of the inductive–recursive (IR) definitions.

Upon reflecting on this possibility, it shall become manifest that the concepts cannot be defined uniformly in A and B either; rather, the A and B must take part in the recursive construction of both the set $\mathcal{E}q(A,B)$ as well as the map $\mathcal{R}el_{A,B}(e)$. Thus, the arguments A and B are to be treated as *indices*, so that we are dealing with an *indexed inductive–recursive definition* (IIRD).

We are now in the position to answer the questions posed above. The notion of equivalence of types A and B and the notion of elements of the corresponding types being related over an equivalence are both defined simultaneously by indexed induction–recursion. The definition appears below.

$$
\begin{aligned}
&\mathsf{Inductive}\ \mathcal{E}q : \mathcal{U} \to \mathcal{U} \to * \ :=\\
&\quad | \ \mathsf{r}(\circledast) \quad : \quad \mathcal{E}q \circledast \circledast \\
&\quad | \ \textcircled{\Pi}^* \{AA' : \mathcal{U}\}\{B : \mathcal{T}A \to \mathcal{U}\}\{B' : \mathcal{T}A' \to \mathcal{U}\} \\
&\qquad (A^* : \mathcal{E}q A A')(B^* : \Pi a{:}\mathcal{T}A \Pi a'{:}\mathcal{T}A' \Pi a^* : \mathcal{R}el\, A^* a a'.\mathcal{E}q(Ba)(B'a')) \\
&\qquad\qquad : \quad \mathcal{E}q(\textcircled{\Pi} AB)(\textcircled{\Pi} A'B') \\
&\quad | \ \textcircled{\Sigma}^* \{AA' : \mathcal{U}\}\{B : \mathcal{T}A \to \mathcal{U}\}\{B' : \mathcal{T}A' \to \mathcal{U}\} \\
&\qquad (A^* : \mathcal{E}q A A')(B^* : \Pi a{:}\mathcal{T}A \Pi a'{:}\mathcal{T}A' \Pi a^* : \mathcal{R}el\, A^* a a'.\mathcal{E}q(Ba)(B'a')) \\
&\qquad\qquad : \quad \mathcal{E}q(\textcircled{\Sigma} AB)(\textcircled{\Sigma} A'B') \\
&\mathsf{with}\quad \mathcal{R}el : \Pi\{A\}\{B\} : \mathcal{U}.\ \mathcal{E}q AB \to \mathcal{T}A \to \mathcal{T}B \to * \\
&\qquad \mathcal{R}el\,(\mathsf{r}(\circledast))AB \quad = \quad \mathcal{E}q AB \\
&\qquad \mathcal{R}el\,(\textcircled{\Pi}^* A^* B^*) f f' \quad = \quad \Pi x{:}\mathcal{T}A \Pi x'{:}\mathcal{T}A' \Pi x^* : \mathcal{R}el\, A^* x x'. \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{R}el\,(B^* x x' x^*)(fx)(f'x') \\
&\qquad \mathcal{R}el\,(\textcircled{\Sigma}^* A^* B^*) p p' \quad = \quad \Sigma x^* : \mathcal{R}el\, A^*(\pi_1 p)(\pi_1 p'). \\
&\qquad\qquad\qquad\qquad\qquad \mathcal{R}el\,(B^*(\pi_1 p)(\pi_1 p')x^*)(\pi_2 p)(\pi_2 p')
\end{aligned}
$$

We denote the system obtained by extending $\lambda{*}\mathcal{U}$ with the above IIRD by $\lambda{*}\mathcal{U}\mathcal{E}$.

We remark that $\lambda{*}\mathcal{U}$ is a subsystem of $\lambda{*}\mathcal{U}\mathcal{E}$ in the sense that every term of $\lambda{*}\mathcal{U}$ is a term of $\lambda{*}\mathcal{U}\mathcal{E}$, and every derivation in $\lambda{*}\mathcal{U}$ is also a derivation in $\lambda{*}\mathcal{U}\mathcal{E}$.

▶ **Definition 9.** Let $(-)' : \mathcal{T}erms(\lambda{*}\mathcal{U}) \to \mathcal{T}erms(\lambda{*}\mathcal{U})$ be the operation of marking every variable with an apostrophe.

The next operation computes the substitution into a term t of a "universal path" in the context of t. The context of t^* is three times larger than t: not only do we add the apostrophized variables, but also the starred variables which witness that x and x' are related.

▶ **Definition 10.**

$(-)^* : \mathcal{T}erms(\lambda{*}\mathcal{U}) \to \mathcal{T}erms(\lambda{*}\mathcal{U}\mathcal{E})$

$$
\begin{aligned}
(x)^* &= x^* \\
\circledast^* &= \mathsf{r}(\circledast) \\
(\textcircled{\Pi} AB)^* &= \textcircled{\Pi}^* A^* B^* \\
(\textcircled{\Sigma} AB)^* &= \textcircled{\Sigma}^* A^* B^* \\
(\lambda x{:}A.b)^* &= \lambda x{:}A\ \lambda x'{:}A'\ \lambda x^* : \mathcal{R}el\, A^* x x'.b^* \\
(fa)^* &= f^* a a' a^* \\
(a,b)^* &= (a^*, b^*) \\
(\pi_1 p)^* &= \pi_1 p^* \\
(\pi_2 p)^* &= \pi_2 p^*
\end{aligned}
$$

▶ **Theorem 11.** $(M[N/x])' = M'[N'/x']$

▶ **Theorem 12.** $(M[N/x])^* = M^*[N/x, N'/x', N^*/x^*]$

Proof.

Axiom. $(\circledast[N/x])^* = (\circledast)^* = \mathsf{r}(\circledast) = \mathsf{r}(\circledast)[N/x, N'/x', N^*/x^*]$

Variable.

$$
(y[N/x])^* = \begin{cases} (x[N/x])^* = N^* = x^*[N/x, N'/x', N^*/x^*] & y = x \\ (y[N/x])^* = y^* = y^*[N/x, N'/x', N^*/x^*] & y \neq x \end{cases}
$$

Product.

$$
\begin{aligned}
(\textcircled{\Pi} AB\ [N/x])^* &= (\textcircled{\Pi} A[N/x] B[N/x])^* \\
&= \textcircled{\Pi}^* (A[N/x])^* (B[N/x])^* \\
&= \textcircled{\Pi}^* A^*[N/x, N'/x', N^*/x^*] B^*[N/x, N'/x', N^*/x^*] \\
&= (\textcircled{\Pi}^* A^* B^*)[N/x, N'/x', N^*/x^*]
\end{aligned}
$$

Sum.

$$
\begin{aligned}
(\textcircled{\Sigma} AB\ [N/x])^* &= (\textcircled{\Sigma} A[N/x] B[N/x])^* \\
&= \textcircled{\Sigma}^* (A[N/x])^* (B[N/x])^* \\
&= \textcircled{\Sigma}^* A^*[N/x, N'/x', N^*/x^*] B^*[N/x, N'/x', N^*/x^*] \\
&= (\textcircled{\Sigma}^* A^* B^*)[N/x, N'/x', N^*/x^*]
\end{aligned}
$$

Abstraction. We remark that the variables can always be chosen so as not to interfere.

$$\begin{aligned}((\lambda y{:}A.b)[N/x])^* &= (\lambda y : A[N/x].b[N/x])^* \\ &= \lambda y{:}A[N/x]\ \lambda y'{:}(A[N/x])'\ \lambda y^* : \mathcal{Rel}\,(A[N/x])^* yy'.(b[N/x])^* \\ &= \lambda y{:}A[N/x]\ \lambda y'{:}A'[N'/x']\ \lambda y^* : \mathcal{Rel}\,A^*[N/x, N'/x', N^*/x^*]yy'. \\ &\qquad\qquad b^*[N/x, N'/x', N^*/x^*] \\ &= \lambda y{:}A[N, N', N^*/x, x', x^*]\ \lambda y'{:}A'[N, N', N^*/x, x', x^*] \\ &\qquad \lambda y^* : (\mathcal{Rel}\,A^* yy')[N, N', N^*/x, x', x^*].b^*[N, N', N^*/x, x', x^*] \\ &= (\lambda y{:}A\ \lambda y'{:}A'\ \lambda y^* : \mathcal{Rel}\,A^* yy'.b^*)[N, N', N^*/x, x', x^*] \\ &= (\lambda y : A.b)^*[N/x, N'/x', N^*/x^*]\end{aligned}$$

Application.

$$\begin{aligned}(st[N/x])^* &= (s[N/x]t[N/x])^* \\ &= (s[N/x])^*(t[N/x])(t[N/x])'(t[N/x])^* \\ &= (s^*[N, N', N^*/x, x', x^*]t[N/x]t'[N'/x]t^*[N, N', N^*/x, x', x^*] \\ &= (s^*tt't^*)[N, N', N^*/x, x', x^*] \\ &= (st)^*[N, N', N^*/x, x', x^*]\end{aligned}$$

Pairing.

$$\begin{aligned}((s, t)[N/x])^* &= (s[N/x], t[N/x])^* \\ &= ((s[N/x])^*, (t[N/x])^*) \\ &= (s^*[N/x, N'/x', N^*/x^*], t^*[N/x, N'/x', N^*/x^*]) \\ &= (s^*, t^*)[N, N', N^*/x, x', x^*] \\ &= (s, t)^*[N, N', N^*/x, x', x^*]\end{aligned}$$

Projection.

$$\begin{aligned}((\pi_i t)[N/x])^* &= (\pi_i t[N/x])^* \\ &= \pi_i(t[N/x])^* \\ &= \pi_i(t^*[N, N', N^*/x, x', x^*]) \\ &= \pi_i t^*[N, N', N^*/x, x', x^*] \\ &= (\pi_i t)^*[N, N', N^*/x, x', x^*]\end{aligned}$$

◀

▶ **Corollary 13.** *Suppose $M = N$. Then $M^* = N^*$.*

Proof. Assume $M = (\lambda x{:}A.s)t$ and $N = s[t/x]$. We have

$$\begin{aligned}M^* = ((\lambda x{:}A.s)t)^* &= (\lambda x{:}A.s)^* tt't^* \\ &= (\lambda x{:}A\ \lambda x'{:}A'\ \lambda x^* : \mathcal{Rel}\,A^* xx'.s^*)tt't^* \\ &= s^*[t/x][t'/x'][t^*/x^*] \\ &= (s[t/x])^* = N^*\end{aligned}$$

where the last equality is by the previous proposition.

Now suppose that $M = \pi_i(t_1, t_2)$, and $N = t_i$. Then

$$M^* = \pi_i(t_1^*, t_2^*) = t_i^* = N^*\,.$$

◀

▶ **Definition 14.** A $\lambda*\mathcal{U}$-context Γ is said to be a $\mathcal{U}$-*context* if Γ is of the form

$$x_1 : \mathcal{T}A_1, \ldots, x_n : \mathcal{T}A_n(x_1, \ldots, x_{n-1})$$

and for $0 \leq i < n$, it holds that

$$x_1 : \mathcal{T}A_1, \ldots, x_i : \mathcal{T}A_i(x_1, \ldots, x_{i-1}) \vdash A_{i+1}(x_1, \ldots, x_i) : \mathcal{U}$$

If Γ is a $\mathcal{U}$-context, and $\Gamma \vdash A : \mathcal{U}$, we call A a $\mathcal{U}$-*type in* Γ.

▶ **Definition 15.** Given a $\mathcal{U}$-context $\Gamma = \{x_1{:}\mathcal{T}A_1, \ldots x_n{:}\mathcal{T}A_n\}$, put

$$\Gamma^* = \begin{cases} x_1 : \mathcal{T}A_1, \ \ldots, \ x_n : \mathcal{T}A_n, \\ x'_1 : \mathcal{T}A'_1, \ \ldots, \ x'_n : \mathcal{T}A'_n, \\ x^*_1 : \mathcal{R}\mathit{el}\, A^*_1 x_1 x'_1, \ \ldots, \ x^*_n : \mathcal{R}\mathit{el}\, A^*_n x_n x'_n \end{cases}$$

Let Γ' be obtained from Γ by apostrophizing every variable, including those occurring in their declared types. Obviously, we can have

▶ **Theorem 16.** $\Gamma \vdash M : A \implies \Gamma' \vdash M' : A'$.

▶ **Theorem 17** (Extensionality of $\lambda*$)**.**

$$\Gamma \vdash_{\lambda*} M : A \quad \Longrightarrow \quad \overline{\Gamma}^* \vdash_{\lambda*\mathcal{U}\mathcal{E}} \overline{M}^* : \mathcal{R}\mathit{el}\, \overline{A}^*\, \overline{M}\, \overline{M}' \tag{14}$$

Proof. We proceed by induction on the derivation.

We will work directly with the images $\overline{M}$ in $\lambda*\mathcal{U}$, confusing M with $\overline{M}$. The distinction is mostly irrelevant, but we need it e.g. in the conversion rule, where the type to be converted to has to be in the image of $\bar{\cdot}$.

Axiom. Suppose $\Gamma \vdash \circledast : \mathcal{T}\circledast$. We have

$$\circledast^* = \mathsf{r}(\circledast) : \mathcal{E}q\circledast\circledast = \mathcal{R}\mathit{el}\, \mathsf{r}(\circledast)\circledast\circledast = \mathcal{R}\mathit{el}\, \circledast^*\circledast\circledast'$$

where $\mathsf{r}(\circledast) : \mathcal{E}q\circledast\circledast$ in any context.

By conversion rule, $\Gamma^* \vdash \circledast^* : \mathcal{R}\mathit{el}\, \circledast^*\circledast\circledast'$.

Variable. Suppose we have a derivation tree with root

$$\frac{\Gamma \vdash A : \mathcal{T}\circledast}{\Gamma, x : \mathcal{T}A \vdash x : \mathcal{T}A}$$

(Notice that the hypothesis says that A is a $\mathcal{U}$-type in Γ.)

By the previous proposition, $\Gamma' \vdash A' : \mathcal{T}\circledast$.

By induction hypothesis, $\Gamma^* \vdash A^* : \mathcal{R}\mathit{el}\, \circledast^* AA'$.

Since $\mathcal{R}\mathit{el}\, \circledast^* AA' = \mathcal{E}qAA'$, we have $\Gamma^* \vdash A^* : \mathcal{E}qAA'$ by conversion.

Yet Γ^* also yields that $\mathcal{T}A : *$ and $\mathcal{T}A' : *$, and thus we may form the context $\Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A' \vdash$. In this context, we may derive that

$$\Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A' \vdash \mathcal{R}\mathit{el}\, A^* x x' : *$$

using the typing rule for the $\mathcal{R}\mathit{el}$ constructor.

By the variable rule, we have

$$\Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A', x^* : \mathcal{R}\mathit{el}\, A^* x x' \vdash x^* : \mathcal{R}\mathit{el}\, A^* x x'$$

The context in the above judgement is $(\Gamma, x : \mathcal{T}A)^*$. The subject is $(x)^*$. The type predicate is as displayed in (14).

Weakening. Suppose they give you

$$\frac{\Gamma \vdash M : \mathcal{T}A \qquad \Gamma \vdash B : \mathcal{T}\circledast}{\Gamma, y : \mathcal{T}B \vdash M : \mathcal{T}A}$$

The induction hypotheses give that

$$\Gamma^* \vdash M^* : \mathcal{R}\mathit{el}\, A^* M M'$$
$$\Gamma^* \vdash B^* : \mathcal{R}\mathit{el}\, \circledast^* B B'$$

As before, we may conclude that $B, B' : \mathcal{U}$ in Γ^*, that $B^* : \mathcal{E}q B B'$, and that $\Gamma^*, y : \mathcal{T}B, y' : \mathcal{T}B'$ is a valid context.
Then $\mathcal{R}\mathit{el}\, B^* y y' : *$, and by weakening we get

$$(\Gamma, y : \mathcal{T}B)^* \vdash M^* : \mathcal{R}\mathit{el}\, A^* M M'$$

Formation. Consider the typing

$$\frac{\Gamma \vdash A : \mathcal{T}\circledast \qquad \Gamma, x{:}\mathcal{T}A \vdash B : \mathcal{T}\circledast}{\Gamma \vdash \textcircled{\Pi} A (\lambda x{:}\mathcal{T}A.B) : \circledast}$$

By induction, $\Gamma^* \vdash A^* : \mathcal{R}\mathit{el}\, \circledast^* A A'$.
By conversion, this gives $\Gamma^* \vdash A^* : \mathcal{E}q A A'$.
We also have $(\Gamma, x{:}\mathcal{T}A)^* \vdash B^* : \mathcal{R}\mathit{el}\, \circledast^* B B'$.
That gives $\Gamma^*, x{:}\mathcal{T}A, x' : \mathcal{T}A', x^* : \mathcal{R}\mathit{el}\, A^* x x' \vdash B^* : \mathcal{E}q B B'$.
Using the abstraction rule, we derive

$$\Gamma^* \vdash \lambda x{:}\mathcal{T}A\ \lambda x'{:}\mathcal{T}A'\ \lambda x^* : \mathcal{R}\mathit{el}\, A^* x x'.B^* : \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A\ \ \Pi x^* : \mathcal{R}\mathit{el}\, A^* x x'.\mathcal{E}q B B'$$

which can be rewritten as

$$\Gamma^* \vdash (\lambda x : \mathcal{T}A.B)^* : \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A\ \ \Pi x^* : \mathcal{R}\mathit{el}\, A^* x x'.\ \mathcal{E}q B B'$$

Using the $\textcircled{\Pi}^*$-constructor, we may derive

$$\Gamma^* \vdash \textcircled{\Pi}^* A^* (\lambda x : \mathcal{T}A.B)^* : \mathcal{E}q (\textcircled{\Pi} A B)(\textcircled{\Pi} A' B')$$

The subject of the above judgment is equal to

$$(\textcircled{\Pi} A (\lambda x : \mathcal{T}A.B))^*$$

while the type is convertible to $\mathcal{R}\mathit{el}\, \mathsf{r}(\circledast)(\textcircled{\Pi} A B)(\textcircled{\Pi} A' B')$. Putting these together using the conversion rule yields

$$\Gamma^* \vdash (\textcircled{\Pi} A (\lambda x : \mathcal{T}A.B))^* : \mathcal{R}\mathit{el}\, \circledast^* (\textcircled{\Pi} A B)(\textcircled{\Pi} A B)'$$

being of the required form.
By replacing Π with Σ, $\textcircled{\Pi}$ with $\textcircled{\Sigma}$, and $\textcircled{\Pi}^*$ with $\textcircled{\Sigma}^*$, we may derive from the same hypotheses that

$$\Gamma^* \vdash (\textcircled{\Sigma} A (\lambda x : \mathcal{T}A.B))^* : \mathcal{R}\mathit{el}\, \circledast^* (\textcircled{\Sigma} A B)(\textcircled{\Sigma} A B)'$$

Abstraction. If we have to do

$$\frac{\Gamma \vdash A : \mathcal{T}\circledast \qquad \Gamma, x : \mathcal{T}A \vdash B : \mathcal{T}\circledast \qquad \Gamma, x : \mathcal{T}A \vdash b : \mathcal{T}B}{\Gamma \vdash \lambda x{:}\mathcal{T}A.b \ :\ \mathcal{T}(\textcircled{\Pi} A (\lambda x{:}\mathcal{T}A.B))}$$

the induction hypotheses yield, with conversion, that

$$\begin{aligned}\Gamma^* &\vdash A^* : \mathcal{E}q\,A\,A'\\ \Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A', x^* : \mathcal{R}el\,A^*xx' &\vdash B^* : \mathcal{E}q\,B\,B'\\ \Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A', x^* : \mathcal{R}el\,A^*xx' &\vdash b^* : \mathcal{R}el\,B^*bb'\end{aligned}$$

Since A, A' are $\mathcal{U}$-types in Γ^*, and $\mathcal{R}el\,A^*xx' : *$, we can apply the abstraction rule three times in a row to see that the context

$$(\Gamma, x : \mathcal{T}A)^* \;=\; \Gamma^*, x : \mathcal{T}A, x' : \mathcal{T}A', x^* : \mathcal{R}el\,A^*xx'$$

yields typing judgment

$$\vdash \lambda x{:}\mathcal{T}A\ \lambda x'{:}\mathcal{T}A'\ \lambda x^* : \mathcal{R}el\,A^*xx'.b^* \;:\; \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\mathcal{R}el\,B^*bb'$$

The subject of this judgment is equal to $(\lambda x : \mathcal{T}A.b)^*$.
The type predicate may be converted as

$$\begin{aligned}&\Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\quad \mathcal{R}el\,B^*bb'\\ =\ &\Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\quad \mathcal{R}el\,B^*((\lambda x{:}\mathcal{T}A.b)x)((\lambda x'{:}\mathcal{T}A'.b')x')\\ =\ &\Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\\ &\qquad\qquad \mathcal{R}el\,((\lambda x : \mathcal{T}A.B)^*xx'x^*)((\lambda x{:}\mathcal{T}A.b)x)((\lambda x'{:}\mathcal{T}A'.b')x')\\ &= \mathcal{R}el\,(\textcircled{\scriptsize D}^*A^*(\lambda x : \mathcal{T}A.B)^*)(\lambda x{:}\mathcal{T}A.b)(\lambda x{:}\mathcal{T}A.b)'\\ &= \mathcal{R}el\,(\textcircled{\scriptsize D}A(\lambda x : \mathcal{T}A.B))^*(\lambda x{:}\mathcal{T}A.b)(\lambda x{:}\mathcal{T}A.b)'\end{aligned}$$

which is of the form (14), as desired.

Application. If the derivation ends with

$$\frac{\begin{array}{ll}\Gamma & \vdash A : \mathcal{T}\circledast\\ \Gamma, x{:}\mathcal{T}A \vdash B : \mathcal{T}\circledast & \Gamma \vdash f : \mathcal{T}(\textcircled{\scriptsize D}A(\lambda x{:}\mathcal{T}A.B)) \qquad \Gamma \vdash a : \mathcal{T}A\end{array}}{\Gamma \vdash fa : B[a/x]}$$

We thus have that that A (A') and B (B') are $\mathcal{U}$-types in Γ (Γ') and $\Gamma, x{:}\mathcal{T}A$ ($\Gamma', x' : \mathcal{T}A'$), respectively.
The induction hypotheses give us

$$\begin{aligned}\Gamma^* &\vdash A^* : \mathcal{E}q\,A\,A'\\ (\Gamma, x : \mathcal{T}A)^* &\vdash B^* : \mathcal{E}q\,B\,B'\\ \Gamma^* &\vdash f^* : \mathcal{R}el\,(\textcircled{\scriptsize D}A(\lambda x{:}\mathcal{T}A.B))^*ff'\\ \Gamma^* &\vdash a^* : \mathcal{R}el\,A^*aa'\end{aligned}$$

We may rewrite the type of f^* as

$$\begin{aligned}&\mathcal{R}el\,(\textcircled{\scriptsize D}A(\lambda x{:}\mathcal{T}A.B))^*ff'\\ &= \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\\ &\qquad \mathcal{R}el\,((\lambda x{:}\mathcal{T}A.B)^*xx'x^*)(fx)(f'x')\\ &= \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\\ &\qquad \mathcal{R}el\,((\lambda x{:}\mathcal{T}A\ \lambda x'{:}\mathcal{T}A'\ \lambda x^* : \mathcal{R}el\,A^*xx'.\ \mathcal{R}el\,B^*)xx'x^*)(fx)(f'x')\\ &= \Pi x{:}\mathcal{T}A\ \Pi x'{:}\mathcal{T}A'\ \Pi x^* : \mathcal{R}el\,A^*xx'.\mathcal{R}el\,B^*(fx)(f'x')\end{aligned}$$

Working in Γ^*, we now apply f^* to a, a', a^* (which types are $\mathcal{T}A, \mathcal{T}A', \mathcal{R}\!el\, A^* aa'$, respectively), in order to obtain

$$f^* aa' a^* : \mathcal{R}\!el\, B^*(fx)(f'x')[a/x, a'/x', a^*/x^*],$$

where we have used the hypotheses on A^* and B^* in validating application typing rule. Since the sets of primed, starred, and vanilla variables are disjoint, and every variable in f' is primed, while every variable in f vanilla, we may rewrite the above as

$$f^* aa' a^* : \mathcal{R}\!el\, B^*[a/x, a'/x', a^*/x^*](fa)(f'a')$$

By the substitution lemma,

$$B^*[a/x, a'/x', a^*/x^*] = (B[a/x])^*$$

We may thus rewrite the above judgment as

$$\Gamma^* \vdash (fa)^* : \mathcal{R}\!el\, B[a/x]^*(fa)(fa)'$$

as required.

Pairing. Given a derivation

$$\frac{\Gamma \vdash A : \mathcal{T}\circledast \qquad \Gamma, x{:}\mathcal{T}A \vdash B : \mathcal{T}\circledast \qquad \Gamma \vdash a : \mathcal{T}A \qquad \Gamma \vdash b : \mathcal{T}B[a/x]}{\Gamma \vdash (a,b) : \mathcal{T}(\textcircled{\Sigma} A(\lambda x : \mathcal{T}A.B))}$$

we have

$$\Gamma^* \vdash a^* : \mathcal{R}\!el\, A^* aa'$$
$$\Gamma^* \vdash b^* : \mathcal{R}\!el\, B[a/x]^* bb'$$

We also have

$$\begin{aligned}
&\mathcal{R}\!el\, (\textcircled{\Sigma} A(\lambda x{:}\mathcal{T}A.B))^*(a,b)(a',b') \\
&= \mathcal{R}\!el\, (\textcircled{\Sigma}^* A^*(\lambda x : \mathcal{T}A.B)^*)(a,b)(a',b') \\
&= \Sigma a^* : \mathcal{R}\!el\, A^* \pi_1(a,b)\pi_1(a',b'). \\
&\qquad \mathcal{R}\!el\, ((\lambda x{:}\mathcal{T}A.B)^* \pi_1(a,b)\pi_1(a',b')a^*)\pi_2(a,b)\pi_2(a',b') \\
&= \Sigma a^* : \mathcal{R}\!el\, A^* aa'.\mathcal{R}\!el\, ((\lambda x{:}\mathcal{T}A.B)^* aa'a^*)bb' \\
&= \Sigma a^* : \mathcal{R}\!el\, A^* aa'.\mathcal{R}\!el\, (B^*[a,a',a^*/x,x',x^*])bb' \\
&= \Sigma a^* : \mathcal{R}\!el\, A^* aa'.\mathcal{R}\!el\, B[a/x]^* bb'
\end{aligned}$$

Using the pairing rule, we see that (a^*, b^*) can be given the type derived above. So by conversion, we find

$$\Gamma^* \vdash (a,b)^* : \mathcal{R}\!el\, (\textcircled{\Sigma} A(\lambda x{:}\mathcal{T}A.B))^*(a,b)(a,b)'$$

as required.

Projections. Given

$$\frac{\Gamma \vdash A : \mathcal{T}\circledast \qquad \Gamma, x{:}\mathcal{T}A \vdash B : \mathcal{T}\circledast \qquad \Gamma \vdash p : \mathcal{T}(\textcircled{\Sigma} A(\lambda x{:}\mathcal{T}A.B))}{\begin{array}{c}\Gamma \vdash \pi_1 p : \mathcal{T}A \\ \Gamma \vdash \pi_2 p : \mathcal{T}B[\pi_1 p/x]\end{array}}$$

we get, by induction hypothesis, that

$$\Gamma^* \vdash p^* : \mathcal{R}el\,(\Sigma A(\lambda x{:}\mathcal{T}A.B))^* pp'$$

By the same computation as the previous case, we see that that the type of p^* above is convertible to

$$\Sigma a^* : \mathcal{R}el\,A^*(\pi_1 p)(\pi_1 p').\mathcal{R}el\,B^*[\pi_1 p, \pi_1 p', a^*/x, x', x^*](\pi_2 p)(\pi_2 p')$$

But then we have

$$\pi_1 p^* : \mathcal{R}el\,A^* \pi_1 p \pi_1 p'$$

$$\pi_2 p^* : \mathcal{R}el\,B^*[\pi_1 p, \pi_1 p', \pi_1 p^*/x, x', x^*](\pi_2 p)(\pi_2 p')$$

The first judgment above already has the form required. As for the second, we use the substitution lemma to rewrite it as

$$(\pi_2 p)^* : \mathcal{R}el\,B[\pi_1 p/x]^*(\pi_2 p)(\pi_2 p)'$$

and this too obeys the form of (14).

Conversion. Suppose

$$\frac{\Gamma \vdash M : A \qquad \Gamma \vdash B : \mathcal{T}\circledast \qquad A = B}{\Gamma \vdash M : B}$$

Since the source derivation is assumed to come from $\lambda*$, we may apply the induction hypothesis and obtain

$$\Gamma \vdash M^* : \mathcal{R}el\,A^* M M'$$

The fact that $A = B$, entails, for deep typographical reasons, that $A' = B'$.
Hence by conversion, we have that $M : B$ as well as $M' : B'$.
But we also have that $B : \mathcal{U}$, so that $B^* : \mathcal{R}el \circledast^* BB'$, or equivalently $B^* : \mathcal{E}qBB'$.
These facts yield that $\mathcal{R}el\,B^* M M' : *$.
By Proposition 13, $\mathcal{R}el\,A^* M M' = \mathcal{R}el\,B^* M M'$.
$\Gamma^* \vdash M^* : \mathcal{R}el\,B^* M M'$ ◀

6 Internalization

Let us summarize the results of the previous two sections.

1. There exists a translation

$$\overline{\cdot} \quad : \quad \mathcal{T}erms(\lambda*) \to \mathcal{T}erms(\lambda*\mathcal{U})$$

such that

$$\Gamma \vdash_{\lambda*} M : A \quad \Longrightarrow \quad \overline{\Gamma} \vdash_{\lambda*\mathcal{U}} \overline{M} : \mathcal{T}\overline{A}$$

2. There exists a translation

$$(\cdot)^* \quad : \quad \mathcal{T}erms(\lambda*\mathcal{U}) \to \mathcal{T}erms(\lambda*\mathcal{U}\mathcal{E})$$

such that

$$\Gamma \vdash_{\lambda*} M : A \quad \Longrightarrow \quad \overline{\Gamma}^* \vdash_{\lambda*\mathcal{U}\mathcal{E}} \overline{M}^* : \mathcal{R}el\,\overline{A}^* \overline{M}\,\overline{M}'$$

As we see, the final transformation

$$M \longmapsto \overline{M}^*$$

does not yet give us what we seek, because it maps a term in system $\lambda*$ to a term in a larger system. For example, the type of $\overline{M}^*$ is, in general, not a type of $\lambda*$, since it may contain references to $\mathcal{U}, \mathcal{T}, \textcircled{\Pi}^*$, etc.

What we are after is an operation that can be iterated. At the very least, it should map terms from one system to the same system. Preferably, the type of the result should also belong to the same universe as the type of the input term.

How can we obtain a universe closed under the $(\mathcal{E}q, \mathcal{R}el)$-family?

Easy: just add the codes corresponding to these types into the grammar of the universe.

$$\begin{aligned}
&\mathsf{Inductive}\ \mathcal{U} : * \ :=\\
&\qquad \mid \circledast \quad : \quad \mathcal{U}\\
&\qquad \mid \textcircled{\Pi} \quad : \quad \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\
&\qquad \mid \textcircled{\Sigma} \quad : \quad \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U}\\
&\qquad \mid \circledcirc \quad : \quad \mathcal{U} \to \mathcal{U} \to \mathcal{U}\\
&\qquad \mid \textcircled{\sim} \quad : \quad \Pi\{AB\} : \mathcal{U}.\ \mathcal{T}[\circledcirc AB] \to \mathcal{T}A \to \mathcal{T}B \to \mathcal{U}
\end{aligned}$$

The decodings of the new symbols are

$$\begin{aligned}
\mathcal{T}(\circledcirc AB) &= \mathcal{E}qAB\\
\mathcal{T}(\textcircled{\sim} ABeab) &= \mathcal{R}el\, eab
\end{aligned}$$

Now, denoting by $\lambda* \mathcal{U}^{\mathcal{E}-}$ the extension of $\lambda*$ with the above universe, we can still define the operation $\overline{M}$ and $(M)^*$, but the problem is that the type of the resulting term

$$\mathcal{R}el\, \overline{A}^* \overline{M}\, \overline{M}' \quad : \quad *$$

belongs to the universe of $\lambda* \mathcal{U}^{\mathcal{E}-}$, and we do not have a reflection from terms of $\lambda* \mathcal{U}^{\mathcal{E}-}$ into $\mathcal{U}$ (since $\mathcal{U}$ does not have an internal universe).

Is it ever possible to define a universe where $(\cdot)^*$ can be iterated?

Surely, one cannot know for sure until one has tried every possible avenue. With the benefit of that knowledge, let us try to expand the realm of inductive-recursive definitions, to allow the decoding function of the IIRD type $\mathcal{E}qAB$, which is itself the decoding of a constructor of a lower index type $\mathcal{U}$ being defined simultaneously with it ($\mathcal{T}(\circledcirc AB) = \mathcal{E}qAB$), to be valued back in the type $\mathcal{U}$! In essence, after evaluating the relation $\mathcal{R}el_{\{A,B\}}(e) : \mathcal{T}A \to \mathcal{T}B \to *$ coded by $e : \mathcal{E}q(A, B)$, we immediately reflect it back into $\mathcal{U}$! We denote $\lambda*$ extended with such a universe by $\lambda* \mathcal{U}^{\mathcal{E}}$.

The two IIRD definitions of the internal universes of $\lambda* \mathcal{U}^{\mathcal{E}-}$ and $\lambda* \mathcal{U}^{\mathcal{E}}$, are given in Figures 2 and 3, and may be compared side-by-side. It shall be seen that the only difference is in the $\mathcal{R}el$ map, and in the value of $\mathcal{T}$ at $\textcircled{\sim}$.

Observe that, this time, for $M : \mathcal{T}A$, the result of applying the $(\cdot)^*$-operator stays in $\mathcal{U}$! Indeed, we have

$$\begin{aligned}
\Gamma \vdash_{\lambda*} M : A \quad &\Longrightarrow \quad \overline{\Gamma} \vdash_{\lambda* \mathcal{U}} \overline{M} : \mathcal{T}\overline{A}\\
&\Longrightarrow \quad \overline{\Gamma}^* \vdash_{\lambda* \mathcal{U}^{\mathcal{E}}} \overline{M}^* : \mathcal{T}(\mathcal{R}el\, \overline{A}^* \overline{M}\, \overline{M}')
\end{aligned}$$

and we have a code in $\mathcal{U}$ for the type of $\overline{M}^*$. This allows us to iterate the $(\cdot)^*$-operator as many times as we like.

$$
\begin{array}{lll}
\mathsf{Inductive}\ \mathcal{U} : * & := & \\
\quad | \circledast & : & \mathcal{U} \\
\quad | \textcircled{\Pi} & : & \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U} \\
\quad | \textcircled{\Sigma} & : & \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U} \\
\quad | \textcircled{\simeq} & : & \mathcal{U} \to \mathcal{U} \to \mathcal{U} \\
\quad | \textcircled{\sim} & : & \Pi\{AB\} : \mathcal{U}.\ \mathcal{T}[\textcircled{\simeq}AB] \to \mathcal{T}A \to \mathcal{T}B \to \mathcal{U} \\
\mathsf{with}\ \mathcal{T} : \mathcal{U} \to * & := & \\
\quad \mathcal{T}(\circledast) & = & \mathcal{U} \\
\quad \mathcal{T}(\textcircled{\Pi}AB) & = & \Pi a : \mathcal{T}A.\ \mathcal{T}[Ba] \\
\quad \mathcal{T}(\textcircled{\Sigma}AB) & = & \Sigma a : \mathcal{T}A.\ \mathcal{T}[Ba] \\
\quad \mathcal{T}(\textcircled{\simeq}AB) & = & \mathcal{E}qAB \\
\quad \mathcal{T}(\textcircled{\sim}eab) & = & \mathcal{R}el\, eab \\
\mathsf{and}\ \mathcal{E}q : \mathcal{U} \to \mathcal{U} \to * & := & \\
\quad | \mathsf{r}(\circledast) & : & \mathcal{E}q \circledast \circledast \\
\quad | \textcircled{\Pi}^* & : & \Pi\{A\}\{A'\}\ \Pi A^* : \mathcal{E}qAA' \\
& & \quad \Pi\{B\}\{B'\}\ \Pi B^* : (\Pi aa'a^*.\ \mathcal{E}q(Ba)(B'a')).\quad \mathcal{E}q(\textcircled{\Pi}AB)(\textcircled{\Pi}A'B') \\
\quad | \textcircled{\Sigma}^* & : & \Pi\{A\}\{A'\}\ \Pi A^* : \mathcal{E}qAA' \\
& & \quad \Pi\{B\}\{B'\}\ \Pi B^* : (\Pi aa'a^*.\ \mathcal{E}q(Ba)(B'a')).\quad \mathcal{E}q(\textcircled{\Sigma}AB)(\textcircled{\Sigma}A'B') \\
\quad | \textcircled{\simeq}^* & : & \Pi\{AA'\}A^*\{BB'\}B^*.\quad \mathcal{E}q(\textcircled{\simeq}AB)(\textcircled{\simeq}A'B') \\
\quad | \textcircled{\sim}^* & : & \Pi AA'A^*BB'B^*ee'e^*aa'a^*bb'b^*.\quad \mathcal{E}q(\textcircled{\sim}eab)(\textcircled{\sim}e'a'b') \\
\mathsf{with}\ \mathcal{R}el\,\{AB : \mathcal{U}\} : \mathcal{E}qAB \to \mathcal{T}A \to \mathcal{T}B \to * & := & \\
\quad | \mathcal{R}el\,(\mathsf{r}(\circledast))AB & = & \mathcal{E}qAB \\
\quad | \mathcal{R}el\,(\textcircled{\Pi}^*A^*B^*)ff' & = & \Pi x{:}\mathcal{T}A\Pi x'{:}\mathcal{T}A'\Pi x^* : \mathcal{R}el\,A^*xx'. \\
& & \qquad \mathcal{R}el\,(B^*xx'x^*)(fx)(f'x') \\
\quad | \mathcal{R}el\,(\textcircled{\Sigma}^*A^*B^*)pp' & = & \Sigma x^* : \mathcal{R}el\,A^*(\pi_1 p)(\pi_1 p'). \\
& & \qquad \mathcal{R}el\,(B^*(\pi_1 p)(\pi_1 p')x^*)(\pi_2 p)(\pi_2 p') \\
\quad | \mathcal{R}el\,(\textcircled{\simeq}^*AA'A^*BB'B^*)ee' & = & \Pi aa'a^*\,\Pi bb'b^*.\ \mathcal{E}q(\textcircled{\sim}eab)(\textcircled{\sim}e'a'b') \\
\quad | \mathcal{R}el\,(\textcircled{\sim}^*AA'A^*BB'B^*ee'e^*aa'a^*bb'b^*)\gamma\gamma' & = & \mathcal{R}el(e^*aa'a^*bb'b^*)\gamma\gamma'
\end{array}
$$

Figure 2 The universe of $\lambda* \mathcal{U}^{\mathcal{E}-}$.

$$
\begin{array}{lll}
\mathsf{Inductive}\ \mathcal{U} : * \ := & & \\
\quad \mid \circledast & : & \mathcal{U} \\
\quad \mid \textcircled{\Pi} & : & \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U} \\
\quad \mid \textcircled{\Sigma} & : & \Pi A : \mathcal{U}.\ (\mathcal{T}A \to \mathcal{U}) \to \mathcal{U} \\
\quad \mid \textcircled{\simeq} & : & \mathcal{U} \to \mathcal{U} \to \mathcal{U} \\
\quad \mid \textcircled{\sim} & : & \Pi\{AB\} : \mathcal{U}.\ \mathcal{T}[\textcircled{\simeq}AB] \to \mathcal{T}A \to \mathcal{T}B \to \mathcal{U} \\
\mathsf{with}\ \ \mathcal{T} : \mathcal{U} \to * \ := & & \\
\quad \mathcal{T}(\circledast) & = & \mathcal{U} \\
\quad \mathcal{T}(\textcircled{\Pi}AB) & = & \Pi a : \mathcal{T}A.\ \mathcal{T}[Ba] \\
\quad \mathcal{T}(\textcircled{\Sigma}AB) & = & \Sigma a : \mathcal{T}A.\ \mathcal{T}[Ba] \\
\quad \mathcal{T}(\textcircled{\simeq}AB) & = & \mathcal{E}qAB \\
\quad \mathcal{T}(\textcircled{\sim}eab) & = & \mathcal{T}(\mathcal{R}\mathit{el}\,eab) \\
\mathsf{and}\ \ \mathcal{E}q : \mathcal{U} \to \mathcal{U} \to * \ := & & \\
\quad \mid \mathsf{r}(\circledast) & : & \mathcal{E}q\circledast\circledast \\
\quad \mid \textcircled{\Pi}^* & : & \Pi\{A\}\{A'\}\,\Pi A^* : \mathcal{E}qAA' \\
 & & \quad \Pi\{B\}\{B'\}\,\Pi B^* : (\Pi aa'a^*.\ \mathcal{E}q(Ba)(B'a')).\quad \mathcal{E}q(\textcircled{\Pi}AB)(\textcircled{\Pi}A'B') \\
\quad \mid \textcircled{\Sigma}^* & : & \Pi\{A\}\{A'\}\,\Pi A^* : \mathcal{E}qAA' \\
 & & \quad \Pi\{B\}\{B'\}\,\Pi B^* : (\Pi aa'a^*.\ \mathcal{E}q(Ba)(B'a')).\quad \mathcal{E}q(\textcircled{\Sigma}AB)(\textcircled{\Sigma}A'B') \\
\quad \mid \textcircled{\simeq}^* & : & \Pi\{AA'\}A^*\{BB'\}B^*.\quad \mathcal{E}q(\textcircled{\simeq}AB)(\textcircled{\simeq}A'B') \\
\quad \mid \textcircled{\sim}^* & : & \Pi AA'A^*BB'B^*ee'e^*aa'a^*bb'b^*.\quad \mathcal{E}q(\textcircled{\sim}eab)(\textcircled{\sim}e'a'b') \\
\mathsf{with}\ \ \mathcal{R}\mathit{el}\,\{AB : \mathcal{U}\} : \mathcal{E}qAB \to \mathcal{T}A \to \mathcal{T}B \to \mathcal{U}\ := & & \\
\quad \mid \mathcal{R}\mathit{el}\,(\mathsf{r}(\circledast))AB & = & \textcircled{\simeq}AB \\
\quad \mid \mathcal{R}\mathit{el}\,(\textcircled{\Pi}^*A^*B^*)ff' & = & \textcircled{\Pi}x{:}A\,\textcircled{\Pi}x'{:}A'\,\textcircled{\Pi}x^* : \mathcal{R}\mathit{el}\,A^*xx'. \\
 & & \quad \textcircled{\sim}\,(B^*xx'x^*)(fx)(f'x') \\
\quad \mid \mathcal{R}\mathit{el}\,(\textcircled{\Sigma}^*A^*B^*)pp' & = & \textcircled{\Sigma}x^* : \mathcal{R}\mathit{el}\,A^*(\pi_1 p)(\pi_1 p'). \\
 & & \quad \textcircled{\sim}\,(B^*(\pi_1 p)(\pi_1 p')x^*)(\pi_2 p)(\pi_2 p') \\
\quad \mid \mathcal{R}\mathit{el}\,(\textcircled{\simeq}^*AA'A^*BB'B^*)ee' & = & \textcircled{\Pi}aa'a^*\ \textcircled{\Pi}bb'b^*.\ \textcircled{\simeq}(\textcircled{\sim}eab)(\textcircled{\sim}e'a'b') \\
\quad \mid \mathcal{R}\mathit{el}\,(\textcircled{\sim}^*AA'A^*BB'B^*ee'e^*aa'a^*bb'b^*)\gamma\gamma' & = & \textcircled{\sim}(e^*aa'a^*bb'b^*)\gamma\gamma'
\end{array}
$$

Figure 3 The universe of $\lambda{*}\mathcal{U}^{\mathcal{E}}$.

▶ Remark. One could naturally wonder whether our use of the expanded induction-recursion format is really necessary, whether it is consistent, and how much logical strength it consumes. Since the logical relation encapsulates some universal properties of the base system, we expect that closing off a universe under its logical relation would require a significant use of logical power. At the same time, we believe that our particular definition could be justified using sufficiently large universes (such as Palmgren's higher-order universes, or Setzer's Mahlo universes), considering the rather obvious inductive structure evident in the form of our definition.

In any case, the above issues do not really concern us here, because we are now ready to define our candidate type system internalizing extensional equality for closed types, and this system is almost certainly consistent. If we ultimately get where we were going, then it's not as important how we got there ...

7 The wormhole

Having defined the universe containing (a reflection of) $\lambda*$ and closed under its own logical relation, we are now ready to ... go there!

We define $\lambda\simeq$, a *dependent type theory with type equality*.

The system is obtained by *unquoting* the internal universe $\mathcal{U}$ of $\lambda* \mathcal{U}^{\mathcal{E}}$.

That is, rather than defining a reflection of meta-level into the object level, we expand our meta-level by *descending into the object level*.

Once we are inside $\mathcal{U}$ we forget that $*$ ever existed,

$$* \quad := \quad \mathcal{U}$$

The syntax of our new environment, as it appears to us, is given in Figure 4. It is almost the same as $\lambda*$ (and we only display the new typing rules), the only differences are:

1. A new type constructor, *type equality*, is present.
2. The universes are stratified to remove inconsistency.

The new type $A \simeq B$ has four introduction rules, stating that every type constructor preserves type equality – including type equality.

It has a single elimination rule which, given an element $e : A \simeq B$, returns a relation

$$\sim e : A \to B \to *$$

For every combination of an introduction rule with the elimination rule, it has a computation rule stating that the relation induced by one of the congruence constructors is given inductively by the logical condition associated to the corresponding type constructor.

▶ **Theorem 18.** *There exists an operation* $(\cdot)^* : \mathcal{T}erms(\lambda\simeq) \to \mathcal{T}erms(\lambda\simeq)$ *such that*

$$\Gamma \vdash_{\lambda\simeq} M : A \implies \Gamma^* \vdash_{\lambda\simeq} M^* : M \sim_{A^*} M'$$

In particular, if $\vdash A : *$ is a closed type, then there exists a closed term

$$\mathsf{r}(A) : A \simeq A$$

We write $a \simeq_A a' := a \sim_{\mathsf{r}(A)} a'$.
If $\vdash a : A$ is a closed term, then there exists closed

$$\mathsf{r}(a) : a \simeq_A a$$

The study of system $\lambda\simeq$ will be continued in future work. A proof of the above theorem may be found in a draft at `http://arxiv.org/abs/1401.1148`.

$$A,t,e \quad ::= \quad *_n \mid x \mid \Pi x{:}A.B \mid \Sigma x{:}A.B \mid A \simeq B \mid a \sim_e b$$
$$\mid \lambda x{:}A.t \mid st \mid (s,t) \mid \pi_1 t \mid \pi_2 t$$
$$\mid *^*_n \mid \Pi^*[x,x',x^*]{:}A^*.B^* \mid \Sigma^*[x,x',x^*]{:}A^*.B^* \mid \simeq^* A^* B^*$$

$$\frac{}{\Gamma \vdash *_n : *_{n+1}} \qquad \frac{\Gamma \vdash A : *_n}{\Gamma \vdash A : *_{n+1}}$$

$$\frac{\Gamma \vdash A : *_n \qquad \Gamma \vdash B : *_n}{\Gamma \vdash A \simeq B : *_n}$$

$$\frac{\Gamma \vdash A : *_n \quad \Gamma \vdash B : *_n \qquad \Gamma \vdash e : A \simeq B}{\Gamma \vdash \ \sim e : A \to B \to *_n}\ (\simeq\text{-Elim})$$

$$(a : A)(b : B) \qquad\qquad a \sim_e b \ :=\ \sim e a b$$

$$\frac{}{\Gamma \vdash *^*_n : *_n \simeq *_n}$$

$$\frac{\left\{\begin{array}{l}\Gamma \vdash A : * \\ \Gamma \vdash A' : * \\ \Gamma \vdash A^* : A \simeq A'\end{array}\right\} \qquad \left\{\begin{array}{r}\Gamma, x : A \ \vdash B : * \\ \Gamma, x' : A' \vdash B' : * \\ \Gamma, x{:}A, x'{:}A', x^* : x \sim_{A^*} x' \vdash B^* : B \simeq B'\end{array}\right\}}{\begin{array}{l}\Gamma \vdash \Pi^*[x,x',x^*] : A^*.\, B^* : \Pi x{:}A.B \simeq \Pi x'{:}A'.B' \\ \Gamma \vdash \Sigma^*[x,x',x^*] : A^*.\, B^* : \Sigma x{:}A.B \simeq \Sigma x'{:}A'.B'\end{array}}$$

$$\frac{\left\{\begin{array}{l}\Gamma \vdash A : * \\ \Gamma \vdash A' : * \\ \Gamma \vdash A^* : A \simeq A'\end{array}\right\} \qquad \left\{\begin{array}{l}\Gamma \vdash B : * \\ \Gamma \vdash B' : * \\ \Gamma \vdash B^* : B \simeq B'\end{array}\right\}}{\Gamma \vdash \simeq^* A^* B^* : (A \simeq B) \simeq (A' \simeq B')}$$

$$\begin{array}{rcl}
A \sim_{*^*} B & \longrightarrow & A \simeq B \\
f \sim_{\Pi^*[x,x',x^*]:A^*.B^*} f' & \longrightarrow & \Pi a{:}A \Pi a'{:}A' \Pi a^* : a \sim_{A^*} a'.\ fx \sim_{B^*[a,a',a^*/x,x',x^*]} f'x' \\
p \sim_{\Sigma^*[x,x',x^*]:A^*.B^*} p' & \longrightarrow & \Sigma a^* : \pi_1 p \sim_{A^*} \pi_1 p'.\ \pi_2 p \sim_{B^*[\pi_1 p, \pi_1 p', a^*/x,x',x^*]} \pi_2 p' \\
e \sim_{\simeq^* A^* B^*} e' & \longrightarrow & \Pi a{:}A \Pi a'{:}A' \Pi a^* : a \sim_{A^*} a' \\
 & & \Pi b{:}B \Pi b'{:}B' \Pi b^* : b \sim_{B^*} b'.\ (a \sim_e b) \simeq (a' \sim_{e'} b')
\end{array}$$

Figure 4 $\lambda\simeq$.

Acknowledgments. I would like to thank Thierry Coquand, Marc Bezem, Henk Barendregt, Jan Willem Klop, Mike Nahas, and Robbert Krebbers for insightful and inspiring discussions about extensionality and related topics. I also thank the anonymous referees for their many suggestions that led to major improvements in this paper.

This research was partially supported by French ANR project COQUAS (12 JS02 006 01).

References

1 Thorsten Altenkirch. Extensional equality in intensional type theory. In *LICS*, pages 412–420. IEEE Computer Society, 1999.

2 Robert Atkey, Neil Ghani, and Patricia Johann. A relationally parametric model of dependent type theory. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL'14, pages 503–515, New York, NY, USA, 2014. ACM.

3 H. P. Barendregt. Lambda calculi with types. In S. Abramsky, Dov M. Gabbay, and S. E. Maibaum, editors, *Handbook of Logic in Computer Science (Vol. 2)*, pages 117–309. Oxford University Press, Inc., New York, NY, USA, 1992.

4 Jean-Philippe Bernardy and Marc Lasson. Realizability and parametricity in pure type systems. In Martin Hofmann, editor, *Foundations of Software Science and Computational Structures*, volume 6604 of *Lecture Notes in Computer Science*, pages 108–122. Springer Berlin Heidelberg, 2011.

5 Thierry Coquand. Equality and dependent type theory, 2011. `http://www.cse.chalmers.se/~coquand/equality.pdf`.

6 Peter Dybjer and Anton Setzer. Indexed induction-recursion. In Reinhard Kahle, Peter Schroeder-Heister, and Robert Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer Berlin Heidelberg, 2001.

7 Neil Ghani, Lorenzo Malatesta, Fredrik Nordvall Forsberg, and Anton Setzer. Fibred data types. In *LICS*, pages 243–252. IEEE Computer Society, 2013.

8 F. Rabe and K. Sojakova. Logical Relations for a Logical Framework. *ACM Transactions on Computational Logic*, 2013. to appear.

9 William W. Tait. Extensional equality in the classical theory of types. In Werner Depauli-Schimanovich, Eckehart Köhler, and Friedrich Stadler, editors, *The Foundational Debate*, volume 3 of *Vienna Circle Institute Yearbook [1995]*, pages 219–234. Springer Netherlands, 1995.

Restricted Positive Quantification Is Not Elementary*

Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz

Institute of Informatics, University of Warsaw, Poland
{alx,urzy,daria}@mimuw.edu.pl

Abstract

We show that a restricted variant of constructive predicate logic with positive (covariant) quantification is of super-elementary complexity. The restriction is to limit the number of eigenvariables used in quantifier introductions rules to a reasonably usable level. This construction suggests that the known non-elementary decision algorithms for positive logic may actually be best possible.

1998 ACM Subject Classification F.4.1 Mathematical Logic, I.2.3 Deduction and Theorem Proving

Keywords and phrases constructive logic, complexity, automata theory

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.251

1 Introduction

Constructive logics are basis for many proof assistants [3, 4, 14, 5] as well as theorem provers [1, 15]. Since these tools are actively used for development of verified software [10, 12] and for formalization of mathematics [8, 9] it is instructive to study computational complexity of decidable fragments of the logics. Especially because first-order intuitionistic logic becomes undecidable at a fairly low level [19].

One such fragment consists of *positive* formulas (understood here as formulas with positive quantification), shown decidable by Mints [13]. As defined there, a formula is positive when it is classically equivalent to one with a quantifier prefix of the form $\forall^*$. If we restrict attention to formulas built with $(\forall, \to)$ only, we can equivalently say that a formula φ is positive if and only if all occurrences of $\forall$ in φ are positive, where:

- The position of $\forall x$ in $\forall x\, \varphi$ is positive;
- Positive/negative positions in φ are respectively positive/negative in $\forall x\, \varphi$ and in $\psi \to \varphi$.
- Positive/negative positions in ψ are respectively negative/positive in $\psi \to \varphi$.

It is not immediate to see that deciding provability for positive formulas is possible. The same positive quantifier may be introduced several times in a proof, and this requires a fresh eigenvariable each time. The number of eigenvariables occurring in a proof is in general unbounded, so the search space for proofs is potentially infinite. However some of the eigenvariables may be regarded as equivalent – variables that "satisfy the same assumptions" can be exchanged with each other. Thus the identity of an eigenvariable x is determined by the set of assumptions made about it. With n assumptions there is 2^n such sets, so we need 2^n eigenvariables. The number of variables to consider grows exponentially at each level of

* Project supported through NCN grant DEC-2012/07/B/ST6/01532.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 251–273

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

nested quantification (see the discussion following Example 6), but altogether it remains finite.

Decision algorithms for formulas of minimal positive logic that rigorously develop the idea sketched above were given by Dowek and Jiang [6, 7], Rummelhoff [16], and Xue and Xuan [22]. It should come as no surprise that these algorithms are of non-elementary complexity (while the analogous problem of satisfiability for $\exists^*$-sentences in classical first-order logic is only NP-complete [2, Thm. 6.4.3]).

As for the lower bound, the best result known up to date is only doubly exponential hardness [18], and our own attempt to prove non-elementary complexity failed; the proof in [17] turned out incorrect.

While the question of an exact lower bound remains open, the contribution of the present paper makes the non-elementary conjecture quite plausible. As noted above, raising the quantifier nesting by one yields at most exponential increase of the number of eigenvariables. This is a crucial argument in the known decidability proofs. We show that if this restriction becomes a part of the problem, i.e., if we require that the number of eigenvariables occurring in proofs is bounded by an appropriate multiply exponential function, then the problem is non-elementary.

This does not necessarily mean that the original problem is non-elementary, as there may be proofs that violate the multiply exponential bound on eigenvariable occurrences, but are easy to find by some algorithm. However, this seems to be very difficult to imagine since then the algorithm would effectively represent a method to compress multiply exponential complicated structures.

Our hardness proof is inspired by an automata-theoretic interpretation of proof-search. The idea is simple and, we believe, quite universal. When attempting to construct a proof of a formula φ, one encounters subproblems of the form $\Gamma \vdash \alpha$. We think of α as if it was a state of an automaton and of Γ as of some kind of memory storage. Applying a proof tactic to $\Gamma \vdash \alpha$, which yields a new proof obligation $\Gamma' \vdash \alpha'$, can be seen as changing the state from α to α' and updating the memory Γ to Γ'. This way, proof construction can simulate a computation of an automaton.

Our *Eden automata* (or "expansible tree automata") are alternating machines operating on data that is structured into *trees of knowledge*. The computation trees of Eden automata correspond directly to proofs (equivalently, λ-terms) and the trees of knowledge represent the structure of binders in proofs. In fact, a slightly more general definition of Eden automata in [18] yields an exact equivalence between proofs and computations. Here, we stick to the weaker version, as we are only interested with a lower bound for the restricted case.

A specific feature of Eden automata is their monotone (non-erasing) access to data, very much as in the works by Leivant or even earlier by Wang [11, 21]. This is so because in a fully-structural logic assumptions are never deleted.

Structure of the paper. Section 2 introduces some notation and states the principal definitions related to logic and lambda-terms used as proof notation. In Section 3 we give some insight into the intricacy of the problem. Then we introduce Eden automata and define the translation of automata into formulas. The main technical development to encode elementary Turing Machines as Eden automata is done in Section 4.

2 Preliminary definitions

We define $\exp_0(n) = n$ and $\exp_{k+1}(n) = 2^{\exp_k(n)}$. A *tree* is a finite partial order $\langle \mathrm{T}, \leq \rangle$ with a least element $\varepsilon_{\mathrm{T}} \in \mathrm{T}$ (the root) and such that every non-root element $w \in \mathrm{T}$ has exactly

one immediate predecessor (parent) v, in which case we say that w is a *child* of v. A *labelled tree* is a function $T : \mathrm{T} \to L$, where T is a tree, and L is a set of labels. We often confuse T with its domain T. If L is a set of m-tuples we may say that the *dimension* of T is m. A *proper ancestor* of a node w in a tree T is either a parent v of w or a proper ancestor of v. (The root of the tree has no proper ancestors.) A node w with exactly h proper ancestors in T is said to be *at depth* h, and then we may write $|w| = h$. The *depth of* T is the maximal depth of a node in T. A tree has *uniform depth* k when all its leaves (maximal elements) are at depth$k$.

It is sometimes convenient to refer to the *level* of a node w which is the depth of the subtree $T_w = \{v \in T \mid w \leq v\}$, rooted at w. An *immediate subtree* of a node w in T is any tree T_v, where v is a child of w.

If $k \in \mathbb{N}$ then $\mathbf{k} = \{0, \ldots, k\}$. If f is any function then $f[x \mapsto a]$ stands for the function f' such that $f'(x) = a$, and $f'(y) = f(y)$, for $y \neq x$. In particular, $T[w \mapsto s]$ is a tree obtained from T by replacing the label at w by s.

Formulas: We consider the *monadic* fragment (all predicates are unary) of first-order intuitionistic logic without function symbols and without equality. Therefore the only object terms are *object variables*, written $x, y, z, \ldots$ For simplicity we only consider two logical connectives: the implication and the universal quantifier. We use standard parentheses-avoiding conventions, in particular we take implication to be right-associative, e.g., $\varphi \to \psi \to \vartheta$ stands for $\varphi \to (\psi \to \vartheta)$.

We deal with *positive formulas*; those are defined in parallel with *negative formulas*:

- An *atom* $P(x)$, where P is a unary predicate symbol and x is an object variable, is both a positive and a negative formula.
- If φ is positive and ψ is negative then $(\varphi \to \psi)$ is a negative formula.
- If φ is negative and ψ is positive then $(\varphi \to \psi)$ is a positive formula.
- If φ is positive and x is an object variable then $(\forall x\, \varphi)$ is a positive formula.
- If φ is negative and x is an object variable then $(\forall x\, \varphi)$ is a negative formula.

The following lemma gives a direct characterization of positive and negative formulas.

▶ **Lemma 1.**

1. *Every positive formula is of the form* $\forall \vec{x}_1(\sigma_1 \to \forall \vec{x}_2(\sigma_2 \to \cdots \to \forall \vec{x}_n(\sigma_n \to \forall \vec{x}_0\, \boldsymbol{a}) \ldots))$, *where* σ_i *are negative, and* $\boldsymbol{a}$ *is an atomic formula.*
2. *Every negative formula is of the form* $\tau_1 \to \tau_2 \to \cdots \to \tau_n \to \boldsymbol{a}$, *where* τ_i *are positive, and* $\boldsymbol{a}$ *is an atomic formula.*

The *rank* of a formula φ, written $rk(\varphi)$, measures the nesting of occurrences of $\forall \vec{x}$ in φ. By induction we define:

- $rk(\mathbf{a}) = 0$, when $\mathbf{a}$ is an atomic formula;
- $rk(\psi \to \vartheta) = \max\{rk(\psi), rk(\vartheta)\}$;
- $rk(\forall x\, \psi) = rk(\psi)$, when ψ begins with $\forall$;
- $rk(\forall x\, \psi) = 1 + rk(\psi)$, otherwise.

Lambda-terms: In addition to object variables, used in formulas, we also have *proof variables* occurring in proofs. We use capital letters, like X, Y, Z, for proof variables and lower case letters, like x, y, z, for object variables.

An *environment* is a set Γ of declarations $(X : \varphi)$, where X is a proof variable and φ is a formula. We often identify Γ with the set of formulas $\{\varphi \mid (X : \varphi) \in \Gamma, \text{ for some } X\}$. A *proof term* (or simply "term") is one of the following:

- a proof variable,
- an abstraction $\lambda X:\varphi.\, M$, where φ is a formula and M is a proof term,
- an abstraction $\lambda x\, M$, where M is a proof term,
- an application MN, where M, N are proof terms,
- an application Mx, where M is a proof term and x is an object variable.

The following type-assignment rules infer judgements of the form $\Gamma \vdash M : \varphi$, where Γ is an environment, M is a term, and φ is a formula. In rule $(\forall\mathcal{I})$ we require $x \notin \mathrm{FV}(\Gamma)$ and y in rule $(\forall\mathcal{E})$ is an arbitrary object variable.

$$\Gamma, X:\varphi \vdash X:\varphi \quad (\mathcal{A}x)$$

$$\frac{\Gamma, X:\varphi \vdash M:\psi}{\Gamma \vdash \lambda X:\varphi.\, M:\varphi\to\psi}\,(\to\mathcal{I}) \qquad \frac{\Gamma \vdash M:\varphi\to\psi \quad \Gamma\vdash N:\varphi}{\Gamma\vdash MN:\psi}\,(\to\mathcal{E})$$

$$\frac{\Gamma\vdash M:\varphi}{\Gamma\vdash \lambda x\, M:\forall x\varphi}\,(\forall\mathcal{I}) \qquad \frac{\Gamma\vdash M:\forall x\varphi}{\Gamma\vdash My:\varphi[x:=y]}\,(\forall\mathcal{E})$$

We may write $\lambda X^{\varphi} M$ for $\lambda X:\varphi.\, M$, and the upper index α in M^{α} means that term M has type α in some (implicit) environment. Other notational conventions are as usual in lambda-calculus, in particular application is left-associative, i.e., MNP stand for $((MN)P)$.

2.1 Restricted proofs and long normal forms

A *redex* is a term of the form $(\lambda x\, M)y$ or of the form $(\lambda Y:\varphi.\, M)N$. A term which does not contain any redex is said to be in *normal form*. It is not difficult to see that normal forms are of the following shapes:

- $XN_1\ldots N_k$, where all N_i are normal forms or object variables;
- $\lambda X:\varphi.\, N$, where N is a normal form;
- $\lambda x\, N$, where N is a normal form.

Normal forms correspond to normal proofs in natural deduction (or to cut-free proofs in sequent calculus). It is known, see e.g., [20, Ch.8], that every well-typed term reduces to one in normal form of the same type. In particular we know that:

If $\Gamma\vdash M:\varphi$ *then there exists a term* N *in normal form with* $\Gamma\vdash N:\varphi$.

Occurrences of a free variable X in a term can be *nested*; this occurs when X is free in some N_i in the context $XN_1\ldots N_k$ where $k\geq 0$. The maximal nesting $\flat(X,M)$ of a variable X in a normal term M is defined formally as:

- $\flat(X,X)=1$, $\flat(X,Y)=0$, when $X\neq Y$;
- $\flat(X, YN_1\ldots N_k) = \flat(X,Y) + \max_i \flat(X,N_i)$;
- $\flat(X,\lambda Y\, N) = \flat(X,N)$, when $X\neq Y$, and $\flat(X,\lambda X\, N)=0$;
- $\flat(X,\lambda y\, N)=\flat(X,N)$.

▶ **Definition 2** (n-restricted proofs)**.** We say that a normal proof M is *n-restricted* when it has the following property: in every subterm of the form $\lambda X:\sigma.\, N$, where $rk(\sigma)=k>0$, the variable X has at most $\exp_k(n)$ nested occurrences in N, i.e., $\flat(X,N)\leq\exp_k(n)$. A judgement is *n-provable* when it has an n-restricted normal proof.

▶ **Problem 3** (restricted decision problem for positive quantification)**.** *Given a positive formula φ and a number n, decide if φ is n-provable.*

The process of proof search is easier to control if we restrict our attention to proofs in *long normal form*.

▶ **Definition 4.** The notion of a term in long normal form (lnf) is defined according to its type in a given environment.

- If N is an lnf of type α then $\lambda x\, N$ is an lnf of type $\forall x\, \alpha$.
- If N is an lnf of type β then $\lambda X : \alpha.\, N$ is an lnf of type $\alpha \to \beta$.
- If $N_1, \ldots, N_n$ are lnf or object variables and $X N_1 \ldots N_n$ is of an atom type then the term $X N_1 \ldots N_n$ is an lnf.

▶ **Lemma 5.** *If $\Gamma \vdash M : \sigma$ and M is in normal form then there exists a long normal form N such that $\Gamma \vdash N : \sigma$. In addition, if M is n-restricted then so is N.*

Proof. First let us define a transformation T which will be used for applications in normal form. In $T^\alpha(M)$ we assume that M is of type α in an appropriate environment; the definition is by induction with respect to α:

- $T^{\forall x.\alpha}(M) = \lambda x\, T^\alpha(Mx)$;
- $T^{\alpha\to\beta}(M) = \lambda X : \alpha.\, T^\beta(MX)$;
- $T^\alpha(M) = M$ if α is an atom type.

Suppose that $M = X N_1 \ldots N_k$, where each N_i is an lnf or an object variable. It is easy to see that if M has type α then $T^\alpha(M)$ is an lnf of type α.

Transformation R takes an argument in normal form and returns its long normal form. In $R^\alpha(M)$ we assume that M is of type α in some environment; the definition is by induction with respect to M:

- $R^{\forall x.\alpha}(\lambda x.P) = \lambda x\, R^\alpha(P)$
- $R^{\alpha\to\beta}(\lambda X : \alpha.P) = \lambda X : \alpha.R^\beta(P)$
- $R^\alpha(X P_1 \ldots P_k) = T^\alpha(X P'_1 \ldots P'_k)$,
  where P'_i is the result of applying R to P_i if P_i is a term, and $P'_i = P_i$ otherwise.

Observe that transformations T and R have the following property:

- They do not change the number and relative position of existing occurrences of free or bound proof variables in a term;
- Whenever a new variable is added, it only occurs once in the result.

The desired term N equals $R^\sigma(M)$. Details are left to the reader. The two properties above ensure that N is n-restricted whenever so is M. ◀

The logic of long normal proofs. We say that a judgement $\Gamma \vdash \varphi$ is *positive* when φ is positive, and all formulas in Γ are negative. The type-assignment rules below preserve positivity, and by Lemma 5 they make a complete proof system for positive judgments.

$$\frac{\Gamma, X : \varphi \vdash M : \psi}{\Gamma \vdash \lambda X : \varphi.\, M : \varphi \to \psi}\,(\to\mathcal{I}) \qquad \frac{\Gamma \vdash M_i : \tau_i, \quad i = 1, \ldots, n}{\Gamma, X : \tau_1 \to \cdots \to \tau_n \to \mathbf{a} \vdash X M_1 \ldots M_n : \mathbf{a}}\,(\to\mathcal{E})$$

$$\frac{\Gamma \vdash M : \varphi}{\Gamma \vdash \lambda x\, M : \forall x \varphi}\,(\forall\mathcal{I})$$

3 Computational content of positive logic

As already mentioned, the main complication of deciding provability of a positive formula is that one quantifier can be introduced several times in a proof and may bring to the derivation several different "eigenvariables". As a result we obtain a potential for unbounded storage. To see how this works let us go through the following example.

▶ **Example 6.** Let $\mathbf{1}$ and $\mathbf{0}$ be unary predicate symbols, and let $\mathsf{G}, \mathsf{L}, \mathsf{U}, \mathsf{Z}$, be nullary atoms.[1] Consider the formulas:

$$\varphi = (\psi \to \mathsf{L}) \to (\mathsf{Z} \to \mathsf{U} \to \mathsf{L}) \to \mathsf{L};$$
$$\psi = \forall x.\, \mathit{Gen}_0(x) \to \mathit{Gen}_1(x) \to \mathit{Zero}(x) \to \mathit{One}(x) \to \mathsf{G},$$

with the following components, whose intended meaning will soon become clear.

$$\begin{aligned} \mathit{Gen}_0(x) &= (\mathbf{0}(x) \to \mathsf{L}) \to \mathsf{G};\\ \mathit{Gen}_1(x) &= (\mathbf{1}(x) \to \mathsf{L}) \to \mathsf{G};\\ \mathit{Zero}(x) &= \mathbf{0}(x) \to \mathsf{Z};\\ \mathit{One}(x) &= \mathbf{1}(x) \to \mathsf{U}. \end{aligned}$$

We show how the process of finding a long normal proof of the formula φ represents a computation of a simple "procedure" consisting of two phases:

1. Nondeterministically generate a number of bits in a loop;
2. Check that there is at least one 0 and at least one 1 among the generated bits.

The atoms correspond to states of the procedure:

- L – the entry point to the main Loop;
- G – Generate a bit;
- Z – test for the presence of Zero;
- U – test for the presence of 1 (a "Unit").

A long normal proof of the formula φ (a lambda term of type φ) must take the shape $\lambda X^{\psi\to\mathsf{L}} \lambda Y^{\mathsf{Z}\to\mathsf{U}\to\mathsf{L}}.\, M$, where the term M of type L should begin either with X or with Y. Let us consider the first possibility, i.e., let $M = XM_1$. The long normal term M_1 has type ψ, so we must have $M_1 = \lambda x_1 \lambda Z_1 U_1 V_1 W_1.\, N_1$, where $Z_1 : \mathit{Gen}_0(x_1)$, $U_1 : \mathit{Gen}_1(x_1)$, $V_1 : \mathit{Zero}(x_1)$, $W_1 : \mathit{One}(x_1)$, and $N_1 : \mathsf{G}$. This way we have replaced the proof goal L by a new proof goal G. We interpret it as passing from state L to state G in a computation.

The term N_1 can now begin with any of the variables Z_1, U_1, V_1, W_1, so let us try Z_1, i.e., take $N_1 = Z_1(\lambda T_1^{\mathbf{0}(x_1)}.\, M_2)$, with M_2 of type L. The variable T_1, which can occur in M_2, is a new assumption of the form $\mathbf{0}(x_1)$ added to the proof environment. Our computational interpretation of this phase is that the bit zero has been just written to the memory cell represented by the eigenvariable x_1 and the control went back to state L.

Asking about M_2 we note that one possibility is $M_2 = X(\lambda x_2 \lambda Z_2 U_2 V_2 W_2.\, N_2)$, with $Z_2 : \mathit{Gen}_0(x_2)$, $U_2 : \mathit{Gen}_1(x_2)$, $V_2 : \mathit{Zero}(x_2)$, $W_2 : \mathit{One}(x_2)$, and $N_2 : \mathsf{G}$. This step introduces to the proof a new eigenvariable x_2 (or allocates a new memory cell x_2). We may now construct $N_2 = Z_2(\lambda T_2^{\mathbf{0}(x_2)}.\, M_3)$, and repeat the loop once more in a slightly different way, by taking $M_3 = X(\lambda x_3 \lambda Z_3 U_3 V_3 W_3.\, U_3(\lambda T_3^{\mathbf{1}(x_3)}.\, M_4))$. Now we have three memory locations x_1, x_2, x_3, containing respectively the values $0, 0, 1$. We could continue in this

[1] Nullary atoms, used for clarity, can be easily replaced by unary ones.

fashion by introducing more locations and more bits, but now we can also complete the proof construction by choosing, for example, $M_4 = Y(V_1(T_1))(W_3(T_3))$. This step represents entering states Z and U, to check the presence of memory locations holding zero and one. Note that this two actions happen independently in parallel (it is a universal computation step). As a result we obtain a complete proof of φ:

$$\lambda X^{\psi\to\mathsf{L}}\lambda Y^{\mathsf{Z}\to\mathsf{U}\to\mathsf{L}}.\, X(\lambda x_1\lambda Z_1U_1V_1W_1.\, Z_1(\lambda T_1^{\mathbf{0}(x_1)}.\, X(\lambda x_2\lambda Z_2U_2V_2W_2.\, Z_2(\lambda T_2^{\mathbf{0}(x_2)}.\, X(\lambda x_3\lambda Z_3U_3V_3W_3.\, Z_3(\lambda T_3^{\mathbf{1}(x_3)}.\, Y(V_1(T_1))(W_3(T_3)))))))).$$

In the above proof, the subterm $V_1(T_1)$ using the variable x_1, can be replaced by $V_2(T_2)$, because assumptions made about x_2 and x_1 are exactly the same. We may say that variables x_1 and x_2 are "equivalent", and from this point of view, introducing x_2 was not necessary. Indeed, the middle line of the above term could simply be deleted without any harm.

As we mentioned before, a proof (for instance a proof of the formula in Example 6) can involve an unbounded number of variables. In [6] it is shown rigorously how some eigenvariables may be eliminated, because "equivalent" variables can replace each other. The term "equivalent" is understood as "satisfying the same assumptions" and a basic instance of such equivalence is presented in Example 6.

The number of necessary non-equivalent eigenvariables is therefore essential to determine the complexity. A closer analysis of the algorithm in [6] reveals a super-elementary (tetration) upper bound, in other words the problem belongs to Grzegorczyk's class E4.

Indeed, a formula of length n has $\mathcal{O}(n)$ different subformulas, so if it only has one quantifier $\forall x$ (like the one in our example) then the number of non-equivalent eigenvariables introduced for the quantifier is (in the worst case) exponential in n, as one has to account for every selection from up to $\mathcal{O}(n)$ subformulas including free occurrences of x. And here the quantifier depth comes into play. Consider a formula of the form $\forall x\,(\ldots\forall y\,\varphi(x,y)\ldots)$. For every eigenvariable x' for $\forall x$ we now have $\mathcal{O}(n)$ subformulas of $\varphi(x',y)$ and therefore up to exponentially many eigenvariables obtained from $\forall y$. Any set of such eigenvariables may potentially be created for a given eigenvariable for $\forall x$, and this gives a doubly exponential number of choices. Two eigenvariables coming from $\forall x$ may be assumed equivalent only when they induce the same choice, so we get a doubly exponential number of possible non-equivalent eigenvariables for $\forall x$. Any additional nested quantifier increases the number of non-equivalent variables exponentially, and this yields the super-elementary upper bound.

3.1 Eden automata

An *Eden automaton* (abbr. Ea) is an alternating computing device, organising its memory into a *tree of knowledge* of bounded depth but potentially unbounded width. The tree initially consists of a single root node and may grow during machine computation, not exceeding a fixed maximum depth. The machine can access memory registers at the presently visited node and its ancestor nodes. This access is limited to using the registers as guards: it can be verified that a flag is up, but checking that a flag is down is simply impossible. Every flag is initially down, but once raised, it so remains forever.

Formally, an Ea is a tuple $\mathcal{A} = \langle\, k, m, \mathcal{R}, Q, q^0, \mathfrak{I}\,\rangle$, where:

- $k \in \mathbb{N}$ is the *depth* of $\mathcal{A}$ (recall the notation $\mathbf{k} = \{0,\ldots,k\}$);
- $\mathcal{R}$ is the finite set of *registers*; the number $m = |\mathcal{R}|$ is the *dimension* of $\mathcal{A}$.
- Q is the finite set of *states*, partitioned as $Q = \bigcup_{i\in\mathbf{k}} Q_i$. In addition, each Q_i splits into disjoint sets $Q_i^\forall$ and $Q_i^\exists$ and we also define $Q^\forall = \bigcup_{i\in\mathbf{k}} Q_i^\forall$ and $Q^\exists = \bigcup_{i\in\mathbf{k}} Q_i^\exists$. States in $Q^\forall$, $Q^\exists$ are respectively *universal* and *existential*.

- $q^0 : \mathbf{k} \to Q$ assigns the *initial state* $q_i^0 \in Q_i$ to every $i \in \mathbf{k}$.
- $\mathfrak{I}$ is the set of *instructions*.

Instructions in $\mathfrak{I}$ *available in state* $q \in Q_i$, may be of the following kinds:

1. "q : `jmp` p", where $p \in Q_j$, and $|i - j| \leq 1$;
2. "q : `check` $R(h)$ `jmp` p", where $p \in Q_i$ and $h \leq i$;
3. "q : `set` $R(h)$ `jmp` p", where $p \in Q_i$ and $h \leq i$;
4. "q : `new`", for $i < k$.

Instructions available in $q \in Q_i^\forall$, for any i, must be of kind (1), with $j = i$. If $q \in Q_h$ in (2) or (3) then we write R instead of $R(h)$. An ID (instantaneous description) of $\mathcal{A}$ is a triple $\langle\, q, T, w \,\rangle$, where q is a state and T is a tree of depth at most k, labelled with elements of $\{0,1\}^{\mathcal{R}}$ (i.e., functions from $\mathcal{R}$ to $\{0,1\}$), called *snakes*. That is, if v is a node of T then $T(v)$ is a snake, and $T(v)(R) \in \{0,1\}$ for any register R. When T is known from the context, we write $R(v)$ for $T(v)(R)$. A snake can be identified with a binary string of length m, for example $\vec{0}$ stands for a snake constantly equal to 0. Finally, the component w is a node of T called the *current apple*. We require that $q \in Q_{|w|}$. That is, the internal state always "knows" the depth of the current apple.

The IDs are classified as *existential* and *universal*, depending on their states. The *initial ID* is $\langle\, q_0^0, T_0, \varepsilon \,\rangle$, where T_0 has only one node ε, the root, labelled with $\vec{0}$ (all flags are down).

An ID $C' = \langle\, p, T', w' \,\rangle$ is a *successor* of $C = \langle\, q, T, w \,\rangle$, when C' is a *result of execution* of an instruction $I \in \mathfrak{I}$ at C. We now define how this may happen. Assume that $q \in Q_i$, and first consider case (1) where $I =$ "q : `jmp` p".

- If $p \in Q_i$ then $C' = \langle\, p, T, w \,\rangle$ is the unique result of execution of I at C. (The machine simply changes its internal state from q to p.)
- If $p \in Q_{i-1}$ then the only possible result is $C' = \langle\, p, T, w' \,\rangle$, where w' is the parent node of w. (The machine moves the apple upward and enters state p.)
- If $p \in Q_{i+1}$ then there may be many results of execution of I, namely all IDs of the form $C' = \langle\, p, T, w' \,\rangle$, where w' is any successor of w in T. (The apple is passed downward to a non-deterministically chosen child w' of w.) In case w is a leaf, there is no result (the instruction cannot be executed).

Let now I be of the form (2), i.e., $I =$"q : `check` $R(h)$ `jmp` p", and let $v \in T$ be the (possibly improper) ancestor of w such that $|v| = h$. If register R at v is 1 (i.e., $T(v)(R) = 1$) then the only result of execution of I at C is $\langle\, p, T, w \,\rangle$. Otherwise there is no result.

If is of the form (3), i.e., $I =$ "q : `set` $R(h)$ `jmp` p" and v is the ancestor of w with $|v| = h$, then the only result of execution of I at C is $C' = \langle\, p, T', w \,\rangle$, where T' is like T, except that in T' the register R at node v is set to 1. That is, $T' = T[v \mapsto T(v)[R \mapsto 1]]$. Observe that it does not matter whether $T(v)(R) = 1$ or $T(v)(R) = 0$.

The last case is (4), i.e., $I =$ "q : `new`" with $i \neq k$. The result of execution of I at C is unique and has the form $C' = \langle\, q_{i+1}^0, T', w' \,\rangle$, where T' is obtained from T by adding a new successor node w' of w, with $T'(w') = \vec{0}$. (The apple goes to the new node and the machine enters the appropriate initial state.)

The semantics of Eas is defined in terms of *eventually accepting* IDs. We say that an existential ID is eventually accepting when *at least one* of its successors is eventually accepting. Dually, a universal ID is eventually accepting when *all* its successors are eventually accepting. Finally we say that an automaton is *eventually accepting* when its initial ID is eventually accepting.

Note that a universal ID with no successors is eventually accepting. By our definition this may only happen when no instruction is available in the appropriate universal state; such states may therefore be called *accepting states*.

A *computation* of an Ea, an alternating machine, should be imagined in the form of a tree of IDs. Every existential node represents a non-deterministic choice and has at most one child. Every universal node has as many children as there are successor IDs. (In other words, a computation represents a strategy in a game.) Such a computation is accepting if every branch ends in a universal leaf.

Restricted computation

The idea of a tree of knowledge is that each node in the tree corresponds directly to an eigenvariable in a proof. Therefore our restriction on proofs gives rise to a restriction for trees: if every child of a node w has at most n children, then the number of children of w should not exceed 2^n. This motivates the following definition. We say that an ID $\langle\, q, T, w \,\rangle$ of an Eden automaton is *n-restricted* when it satisfies the following condition:

- Every node w of T which is at level $i > 0$ has at most $\exp_i(n)$ children.

We are interested in *n-restricted computations*, where all IDs are n-restricted. More formally, we say that an ID is *eventually n-accepting* if it is n-restricted, and

- either it is existential and it has an eventually n-accepting successor,
- or it is universal and all its successors are eventually n-accepting.

3.2 The encoding

Throughout this section we assume that the parameter n is fixed. Our goal is to encode an Ea with a positive first-order formula in such a way that the automaton has an accepting n-restricted computation if and only if the formula has an n-restricted normal proof. Given an automaton $\mathcal{A} = \langle\, k, m, \mathcal{R}, Q, q^0, \mathfrak{I} \,\rangle$, our formula uses unary predicate symbols q, for all $q \in Q$, and R, for all $R \in \mathcal{R}$. Each individual variable is of the form x_i or x_i^w, where $i \in \mathbf{k}$ and w is a node in some tree of knowledge. For a root node ε, we identify x_0^ε with x_0.

Notation: If S is a set of formulas $\{\alpha_1, \ldots, \alpha_k\}$ then $S \to \beta$ abbreviates the formula $\alpha_1 \to \cdots \to \alpha_k \to \beta$. Similarly $\lambda X^S.\, M$ and $\lambda \vec{X} : S.\, M$ abbreviate $\lambda X_1^{\alpha_1} \ldots X_k^{\alpha_k}.M$.

Convention: Without loss of generality we can assume that for every $i < k$ there is exactly one state $q \in Q_i$ such that the instruction "q : `new`" belongs to $\mathfrak{I}$. Indeed, otherwise we can modify the automaton by adding designated "transfer states" $q_i^\star$ to Q_i and replacing each "q : `new`" by "q : `jmp` $q_i^\star$" and "$q_i^\star$: `new`" when necessary.

Encoding instructions

For every $i \in \mathbf{k}$, we define a set of formulas S_i. With one exception (downward moves), formulas in S_i represent instructions available in states $q \in Q_i$. The definition is by backward induction with respect to i.

Universal states: Let $q \in Q_i^\forall$, and let "q : `jmp` p_1", ..., "q : `jmp` p_r" be all the instructions available in q. Then the following formula belongs to S_i:

$$p_1(x_i) \to \cdots \to p_r(x_i) \to q(x_i).$$

Existential states (downward moves): For every instruction of the form "q : `jmp` p", where $q \in Q_{i-1}$ and $p \in Q_i$, the following formula belongs to S_i:

$$p(x_i) \to q(x_{i-1}).$$

In this case the instruction is executed at depth $i-1$, but the formula is in S_i.

Existential states (other moves): Let now $q \in Q_i^{\exists}$. For each of the following instructions available in q, there is one formula in S_i:

- For "q : `jmp` p", where $p \in Q_j$ and $j \in \{i, i-1\}$, the formula is $p(x_j) \to q(x_i)$.
- For "q : `check` $R(h)$ `jmp` p", the formula is $p(x_i) \to R(x_h) \to q(x_i)$.
- For "q : `set` $R(h)$ `jmp` p", the formula is $(R(x_h) \to p(x_i)) \to q(x_i)$.
- For "q : `new`", the formula is $\forall x_{i+1}(S_{i+1} \to q_{i+1}^0(x_{i+1})) \to q(x_i)$.

The set of formulas S_i contains only one copy of S_{i+1} (state q_{i+1}^0 is fixed and by our convention so is q), whence the size of S_0 is polynomial in the size of $\mathcal{A}$. It is also worth pointing out that the rank of all the above formulas is zero, with the exception of the formula for "q : `new`", the rank of the latter is $k-i$ when $q \in Q_i$ (note that $i < k$).

The number of nested occurrences of a variable $Z : \forall x_{i+1}(S_{i+1} \to q_{i+1}^0(x_{i+1})) \to q(x_i)$ exactly corresponds to the number of different eigenvariables induced by the quantifier $\forall x_{i+1}$. Indeed, Z occurs in contexts of the form "$Z(\lambda x_{i+1} \dots Z(\lambda x'_{i+1} \dots Z(\lambda x''_{i+1}.\, M)\dots)\dots)$", and all the individual variables x_{i+1}, x'_{i+1}, x''_{i+1}, … may be free inside M.

Encoding IDs

Let now S be a set of formulas and let w be a node of depth i in a tree of knowledge. For every $j \le i$, replace all occurrences of x_j in S by x_j^v, where v is an ancestor of w of depth j. The result is denoted by $S[w]$, and is formally defined by induction with respect to $|w|$:

$$S[w] = \begin{cases} S, & \text{if } w = \varepsilon; \\ S[v][x_{|w|} := x_{|w|}^w], & \text{if } w \text{ is a child of } v. \end{cases}$$

For a given tree of knowledge T, we define sets of formulas:

$$\Gamma_T^R = \{R(x_i^w) \mid w \in T \wedge |w| = i \wedge T(w)(R) = 1\};$$
$$\Gamma_T^S = \bigcup\{S_i[w] \mid w \in T \wedge |w| = i\};$$
$$\Gamma_T = \Gamma_T^R \cup \Gamma_T^S$$

where S_i is as defined above. Note that $\mathrm{FV}(\Gamma_T) = \{x_i^w \mid w \in T \wedge |w| = i\}$.

The following lemma reduces the halting problem for n-restricted computations of Eas to n-restricted provability of positive formulas. In order to state it in a form permitting a proof by induction we need to refine the definition of n-restricted proof to take care of free assumptions. This is done with the following notion of a proof that *respects a tree of knowledge*.

An environment of the form Γ_T contains, for every non-leaf node $w \in T$, a declaration

$$Z_w : \forall x_{i+1}(S_{i+1}[w] \to q_{i+1}^0(x_{i+1})) \to q(x_i^w),$$

where i is the depth of w. Now for every child v of w there is a variable x_{i+1}^v in $\mathrm{FV}(\Gamma_T)$. These eigenvariables should be thought of as reducing the limit of nested occurrences of Z_w in proofs defined in Γ_T. Let ch_w^T be the number of children of w in T. We say that a proof $\Gamma_T \vdash M : q(x_i)$ *respects* tree T if $\flat(Z_w, M) \le \exp_{k-i}(n) - ch_w^T$, for every $i \le k$ and every node w at depth i.

▶ **Lemma 7.** *Let $\mathcal{A}$ be an Eden automaton. An ID of $\mathcal{A}$ of the form $\langle\, q, T, w \,\rangle$ is eventually n-accepting if and only if the positive judgement $\Gamma_T \vdash q(x^w_{|w|})$ has an n-restricted long normal proof that respects T.*

In particular, the initial ID is eventually n-accepting if and only if $\vdash \Gamma^S_{T_0} \to q^0_0(x_0)$, where T_0 is the initial tree of knowledge, has an n-restricted long normal proof.

Proof. ($\Rightarrow$) Let $\mathcal{A}$ be an Eden automaton and let $\langle\, q, T, w \,\rangle$ be an eventually n-accepting ID of $\mathcal{A}$. We will show that $\Gamma_T \vdash q(x^w_i)$, where $i = |w|$, has an n-restricted proof that respects T. We proceed by induction with respect to the definition of eventually n-accepting IDs.

If q is a universal state and $\langle\, q, T, w \,\rangle$ is eventually n-accepting then all successors of $\langle\, q, T, w \,\rangle$ are eventually n-accepting. Every successor ID corresponds to some instruction "$q : \texttt{jmp}\ p_j$" for $j = 1, \ldots, s$. By the induction hypothesis we have $\Gamma_T \vdash p_j(x^w_i)$ for $j = 1, \ldots, s$.

By the definition of Γ_T, the formula $p_1(x_i) \to \cdots \to p_s(x_i) \to q(x_i)$ belongs to S_i and $p_1(x^w_i) \to \cdots \to p_s(x^w_i) \to q(x^w_i)$ belongs to $S_i[w]$. Since $S_i[w] \subseteq \Gamma_T$, it follows that $\Gamma_T \vdash q(x^w_i)$.

If q is an existential state and $\langle\, q, T, w \,\rangle$ is eventually n-accepting then there exists a successor of $\langle\, q, T, w \,\rangle$ which is eventually n-accepting. This successor $\langle\, p, T', w' \,\rangle$ is a result of execution of an instruction I of $\mathcal{A}$, applicable in state q. We check the possible forms of I.

If I is "$q : \texttt{jmp}\ p$", where $q \in Q_i$, $p \in Q_j$, one has $T' = T$ and either $w = w'$ or w' is an immediate predecessor or successor of$w$ in T. By the induction hypothesis we have $\Gamma_T \vdash p(x^{w'}_j)$. Since Γ_T contains the formula $p(x^{w'}_j) \to q(x^w_i)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

For "$q : \texttt{check}\ R(j)\ \texttt{jmp}\ p$", where $p, q \in Q_i$, let v be the ancestor of w in T such that $|v| = j$. One has $T' = T$, $w' = w$ and the register R at v is set to 1 (since otherwise this instruction cannot be executed). By the induction hypothesis, $\Gamma_T \vdash p(x^w_i)$. Since Γ_T contains the formula $p(x^w_i) \to R(x^v_j) \to q(x^w_i)$ and the atom $R(x^v_j)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

For "$q : \texttt{set}\ R(j)\ \texttt{jmp}\ p$", where $p, q \in Q_i$, let v be the ancestor of w in T such that $|v| = j$. One has $w' = w$ and $T' = T[v \mapsto T(v)[R \mapsto 1]]$. By the induction hypothesis we have $\Gamma_{T'} \vdash p(x^w_i)$. Note that $\Gamma_{T'} = \Gamma_T \cup R(x^v_j)$, and consequently $\Gamma_T \vdash R(x^v_j) \to p(x^w_i)$. Since Γ_T contains the formula $(R(x^v_j) \to p(x^w_i)) \to q(x^w_i)$, we conclude that $\Gamma_T \vdash q(x^w_i)$.

In all the above cases, the assumptions used in the appropriate proof steps are formulas of rank rk equal to zero. Therefore it follows immediately from the induction hypothesis that the obtained proofs are n-restricted and respect T. These proofs are also long normal, as all are of the form $X\vec{N}$, where $\vec{N}$ are long normal by induction.

Only the last case involves quantification. For "$q : \texttt{new}$", where $q \in Q_i$, $p = q^0_{i+1}$, the tree T' is obtained from T by adding a brand new child w' of w labelled $\vec{0}$ (empty registers). From the induction hypothesis we know that $\Gamma_{T'} \vdash M : q^0_{i+1}(x^{w'}_{i+1})$ where M respects T'. Note that $\Gamma_{T'} = \Gamma_T \cup S_{i+1}[w']$, so we may deduce that $\Gamma_T \vdash \lambda\vec{X}^{S_{i+1}[w']}.\, M : S_{i+1}[w'] \to q^0_{i+1}(x^{w'}_{i+1})$. The variable $x^{w'}_{i+1}$ does not appear in Γ_T, hence we also have

$$\Gamma_T \vdash \lambda x_{i+1}.\lambda\vec{X} : S_{i+1}[w'][x^{w'}_{i+1} := x_{i+1}].\, M : \forall x_{i+1}(S_{i+1}[w'][x^{w'}_{i+1} := x_{i+1}] \to q^0_{i+1}(x_{i+1})).$$

Since $S_{i+1}[w'][x^{w'}_{i+1} := x_{i+1}] = S_{i+1}[w]$ and Γ_T contains the declaration

$$Z_w : \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1})) \to q_i(x^w_i),$$

we conclude that $\Gamma_T \vdash Z_w(\lambda x_{i+1}.\lambda\vec{X}^{S_{i+1}[w']}.M) : q_i(x^w_i)$. This is a long normal proof introducing a single application of the proof variable Z_w. It respects T because M respects T' and the number of children of w in T is smaller by one than the number in T'. Also the obtained proof is n-restricted, because so is M and because M respects T', in particular the number of nested occurrences of $Z_{w'}$ in M is at most $exp_{k-i-1}(n)$ (node w' has no children).

($\Leftarrow$) Suppose that $\langle\, q, T, w \,\rangle$ is an ID of an automaton $\mathcal{A}$ such that $\Gamma_T \vdash N : q(x_i^w)$, where $i = |w|$ and where N is an n-restricted long normal form that respects T. We show, by induction with respect to N, that $\langle\, q, T, w \,\rangle$ is eventually n-accepting. Since $q(x_i^w)$ is an atom, we must have $N = XN_1 \dots N_r$, for some X and some long normal forms $N_1, \dots, N_r$. In addition, there must be a declaration $(X : \varphi) \in \Gamma_T$, where $\varphi = \tau_1 \to \dots \to \tau_r \to q(x_i^w)$, and $\Gamma_T \vdash N_l : \tau_l$, for each l.

Let $q \in Q^i_\forall$ and let "q : `jmp` p_j", for $j = 1, \dots, s$, be all instructions available in state q. By the definition of Γ_T, there is only one formula φ that ends with the atom $q(x_i^w)$, namely $\varphi = p_1(x_i^w) \to \dots \to p_s(x_i^w) \to q(x_i^w)$. Therefore, $r = s$ and for every $j = 1, \dots, r$, we have $\Gamma_T \vdash N_j : p_j(x_i^w)$. By the induction hypothesis we know that $\langle\, p_j, T, w \,\rangle$ are eventually n-accepting. Since a universal ID is eventually n-accepting when all its successors are eventually n-accepting, we get the desired conclusion.

Let $q \in Q^i_\exists$. Since the formula φ ends with $q(x_i^w)$, it must correspond to some instruction I that is available in state q. We need to show that I can be executed and that a result of execution of I is eventually n-accepting. This will imply that also $\langle\, q, T, w \,\rangle$ is eventually n-accepting.

If φ has the form $p(x_j^{w'}) \to q(x_i^w)$, for some variable $x_j^{w'}$, then I is "q : `jmp` p". Note that such a φ may occur in Γ_T only when w' is a node of T, more precisely, node w' is either w or it is an immediate predecessor or successor of w in T. By the induction hypothesis applied to $\Gamma_T \vdash N_1 : p(x_j^{w'})$, we conclude that $\langle\, p, T, w' \,\rangle$ is eventually n-accepting.

If φ is $p(x_i^w) \to R(x_j^v) \to q(x_i^w)$ then I is "q : `check` $R(j)$ `jmp` p". We need to show that I can be executed, i.e., that $T(v)(R) = 1$ where v is the ancestor of w in T with $|v| = j$. We know that $\Gamma_T \vdash N_1 : p(x_i^w)$ and $\Gamma_T \vdash N_2 : R(x_j^v)$. The only formula in Γ_T of the form $\alpha_1 \to \dots \to \alpha_k \to R(x_j^v)$ is $R(x_j^v)$. By the definition of Γ_T, if $R(x_j^v) \in \Gamma_T$ then $T(v)(R) = 1$. Hence I can be executed. Since $\Gamma_T \vdash N_1 : p(x_i^w)$, by the induction hypothesis, $\langle\, p, T, w \,\rangle$ (the result of execution of I at $\langle\, p, T, w \,\rangle$) is eventually accepting.

If φ is $(R(x_j^v) \to p(x_i^w)) \to q(x_i^w)$ then I is "q : `set` $R(j)$ `jmp` p" and $j \leq i$. The result of execution of I at $\langle\, q, T, w \,\rangle$ is $\langle\, p, T', w \,\rangle$, where $T' = T[v \mapsto T(v)[R \mapsto 1]]$. We know that $\Gamma_T \vdash N_1 : R(x_j^v) \to p(x_i^w)$. Since N_1 is an lnf, there exists N_1' such that $\Gamma_T, Y : R(x_j^v) \vdash N_1' : p(x_i^w)$. By the induction hypothesis, $\langle\, p, T', w \,\rangle$ is eventually accepting. The last case is when $\varphi = \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1})) \to q(x_i^w)$ is the type of Z_w and the instruction I is "q : `new`". We have $\Gamma_T \vdash Z_w N_1 : q(x_i^w)$ and we also know that $\Gamma_T \vdash N_1 : \forall x_{i+1}(S_{i+1}[w] \to q^0_{i+1}(x_{i+1}))$. Since N_1 is an lnf, it must have the form $N_1 = \lambda x_{i+1} \lambda \vec{Y} : S_{i+1}[w].N_1'$, for some lnf N_1'. Substituting $x_{i+1}^{w'}$ for x_{i+1} we obtain the type assignment

$$\Gamma_T, \vec{Y} : S_{i+1}[w][x_{i+1} := x_{i+1}^{w'}] \vdash N_1'[x_{i+1} := x_{i+1}^{w'}] : q^0_{i+1}(x_{i+1}^{w'}).$$

Note that $S_{i+1}[w][x_{i+1} := x_{i+1}^{w'}]$ equals $S_{i+1}[w']$ and $\Gamma_T, \vec{Y} : S_{i+1}[w'] = \Gamma_{T'}$, where T' is obtained from T by adding a new child w' of w labelled $\vec{0}$. The term $N_1'[x_{i+1} := x_{i+1}^{w'}]$ is n-restricted and respects T' because the top occurrence of Z_w was eliminated, and because w' has no children in T'. Hence, by the induction hypothesis, the result $\langle\, q^0_{i+1}, T', w' \,\rangle$ of execution of I is eventually n-accepting. ◀

4 Eden programming

We begin with a few examples demonstrating how Eden automata can be used to solve computational tasks. They present some techniques exploited in the hardness proof to follow and introduce the reader to the "pseudo-code" we use.

The access to knowledge in an Eden automaton is restricted in that it precludes the possibility to verify that a given bit is 0. This can be partly overcome by a simple trick: use two bits to encode one, 10 for 0 and 01 for 1. This works as long as one can ensure that the two flags are never raised together.

▶ **Example 8.** To be more specific, if we fix 6 registers $L_1, R_1, L_2, R_2, L_3, R_3$ then any word of length 3 can be represented by a snake where exactly one register in each pair L_i, R_i is set to 1. For example, 101 is encoded by $R_1 = L_2 = R_3 = 1$ and $L_1 = R_2 = L_3 = 0$.

Consider an automaton $\mathcal{A}$ of depth 1, with $q_0^0 = q_0$, $q_1^0 = q_1$, and with the instructions (where $q_0 \in Q_0^\exists$, and other states are in $Q_1^\exists$):

q_0 : `new`;

q_1 : `set` $L_1(1)$ `jmp` q_2;	q_2 : `set` $L_2(1)$ `jmp` q_3;	q_3 : `set` $L_3(1)$ `jmp` q_4;
q_1 : `set` $R_1(1)$ `jmp` q_2;	q_2 : `set` $R_2(1)$ `jmp` q_3;	q_3 : `set` $R_3(1)$ `jmp` q_4.

The automaton $\mathcal{A}$ starts in the initial ID in state q_0 with a root-only tree of knowledge. It creates an additional node **d**, a successor of the root, and enters state q_1 at node **d**. The procedure from state q_1 to state q_4 constitutes a for loop, informally written as follows:

$$q_1 : \text{for } i = 1 \text{ to } 3 \text{ do } [\texttt{set } L_i \text{ OR } \texttt{set } R_i]; \text{ goto } q_4.$$

The computation of our automaton has one branch, which ends in an ID where the only child of the root represents a non-deterministically generated word of length 3. The apple is at the child node and the machine is in state q_4.

We can now compose the automaton with another one, $\mathcal{A}'$, which runs after $\mathcal{A}$, i.e., it commences in state q_4. Among its states, $q_1', q_2', q_3', q_{\text{acc}}$ are in $Q_1^\forall$ and other states are in $Q_1^\exists$.

q_4 : `jmp` q_1';

q_1' : `jmp` q_1^{chk};	q_2' : `jmp` q_2^{chk};	q_3' : `jmp` q_3^{chk};
q_1' : `jmp` q_2';	q_2' : `jmp` q_3';	q_3' : `jmp` q_5;
q_1^{chk} : `check` $L_1(1)$ `jmp` q_{acc};	q_2^{chk} : `check` $L_2(1)$ `jmp` q_{acc};	q_3^{chk} : `check` $L_3(1)$ `jmp` q_{acc};

The automaton $\mathcal{A}'$ is initiated in state q_4 in node **d**, a successor of the root. At node **d**, one register in each of the pairs L_1, R_1; L_2, R_2; L_3, R_3 is set to 1. The automaton then enters state q_1' at node **d**. The procedure from state q_1' to state q_5 constitutes a universal for loop, informally written as follows:

$$q_1 : \text{for } i = 1 \text{ to } 3 \text{ do } [\texttt{check } L_i \text{ AND continue}]; \text{ goto } q_5.$$

In successful circumstances, the computation has 4 branches. Three of them end in an accepting ID in the state q_{acc} and the fourth one ends in an ID where all L_1, L_2, L_3 are set to encode the sequence of bits 000. The apple is at the child node and the machine is in state q_5.

These two automata may be viewed as procedures in a single program. The first procedure generates non-deterministically a string of three bits and the second works like a finite automaton that checks if all the bits are equal to 0. (Note that any loop-free finite automaton can be simulated this way.)

4.1 Procedures

Throughout this section we assume that the parameter n is fixed, and we only consider n-restricted computations (a computation which is not n-restricted is illegal). We show how to deal with numbers up to $\exp_k(n)$ using trees of knowledge of depth k.

The trees and automata we consider here have dimension $2n + 7$. We think of the snakes as containing the following parts:

- a *base segment* consisting of $2n$ registers $L_0, R_0, \ldots, L_{n-1}, R_{n-1}$;
- *data* registers: A_0, A_1;
- a *global* register *Steady*;
- *local* registers: *New, Old, Done, Gone.*

The base segment is capable to encode a binary word of length n, using the "two for one" trick, as demonstrated in Example 8. The data registers may contain a binary *value* of a node in a similar way: for $i = 0, 1$, register A_i set to 1 represents the bit i.

We identify binary words of length $\exp_k(n)$ with numbers from 0 to $\exp_{k+1}(n) - 1$, and we use trees of uniform depth k to encode such words-numbers. Informally, the idea is as follows: a word $a_0a_1 \ldots a_{r-1}$ of length r can be represented as the set of pairs $\{(0, a_0), (1, a_1), \ldots, (r-1, a_{r-1})\}$. If a tree T encodes a number i and has value a_i at the root then T represents a pair (i, a_i). A word $a_0a_1 \ldots a_{r-1}$ of length r can thus be encoded by a tree consisting of a root node and a number of immediate subtrees representing the pairs (i, a_i). (Observe that i is then encoded by a string of length of order $\log r$.) Once we know how to encode binary words of length d we can interpret them as numbers from 0 to $2^d - 1$, and use the above method to give an encoding for words of length 2^d.

More precisely, we define what it means that a tree T of uniform depth k *encodes* a word w of length $\exp_k(n)$. To begin with $k = 0$, a tree T consisting of a single node $\mathbf{d}$ *encodes* a binary word $x_0x_1 \ldots x_{n-1}$ of length n when, for each number $i = 1, \ldots, n-1$, we have $T(\mathbf{d})(L_i) = 1$ iff $x_i = 0$, and $T(\mathbf{d})(R_i) = 1$ iff $x_i = 1$. (Note that there are other registers as well, so many trees encode the same number.) A tree T of uniform depth k *encodes* a word $w = x_0x_1 \ldots x_{r-1}$ of length $r = \exp_{k+1}(n)$ when

- T has exactly r immediate subtrees, each encoding a different number $i \in \{0, \ldots, r-1\}$;
- If $\mathbf{d}$ is the root of an immediate subtree encoding i then $T(\mathbf{d})(A_j) = 1$ iff $x_i = j$. (We say that i is the *address* of $\mathbf{d}$ and j is called the *value* of $\mathbf{d}$.)

A node $\mathbf{d}$ in a tree is said to *encode* a word when the subtree rooted at $\mathbf{d}$ encodes that word.

▶ **Remark 9.** One can easily generalize the above definition to words over any l-element finite alphabet $\Sigma = \{a_0, a_1, \ldots, a_{l-1}\}$ with trees of dimension $l \cdot n + l + 5$, and data registers $A_0, \ldots, A_{l-1}$ to represent symbols $a_0, a_1, \ldots, a_{l-1}$.

We now show how Eden automata can manipulate binary words. The automata defined in this section should more adequately be called "procedures" as they are used as subroutines in our main construction. Each procedure is initiated at some specific *start IDs* which are expected to satisfy certain conditions.

We say that a computation initiated in a start ID is called a *successful computation* of a procedure if every branch either ends in an accepting ID, or in an *end ID* (an ID with a specified *end state*), where the control should be passed to another subroutine. In general there may be many occurrences of end IDs in a computation. However, the procedures we consider in this paper have this particular property that every computation contains at most one end ID.

For every k and every $l > k$, we define procedures $\mathcal{M}_k$, $\mathcal{E}_k^l$, S_k, $\mathcal{C}_k^0$, and $\mathcal{C}_k^1$, by simultaneous induction with respect to k. For each of these procedures we first define start and end IDs and formulate the appropriate induction hypothesis in the form of an input-output condition.

Induction hypotheses

Making a new word

For every $k \geq 0$ we define a procedure $\mathcal{M}_k$ to make new words.

Start ID: The current apple is a leaf $\mathbf{d}$ of the tree of knowledge, the snake at $\mathbf{d}$ is empty.
Claim:

1. No computation of $\mathcal{M}_k$ ever uses (jumps, writes to or reads from registers at) any proper ancestor of $\mathbf{d}$.
2. A successful restricted computation of $\mathcal{M}_k$ has only one end ID. At the end ID the apple is back at $\mathbf{d}$, but $\mathbf{d}$ is now a root of a subtree of uniform depth k and $\mathbf{d}$ encodes a non-deterministically chosen word w of length $\exp_k(n)$. All local registers are empty in the subtree rooted at $\mathbf{d}$.

Part 1 of the induction hypothesis is a separation condition which states that the procedure $\mathcal{M}_k$ does not have side effects. This is necessary since the procedure has end states, and computations continues after these are reached.

For the other subroutines we define no end IDs and no similar separation conditions; their only purpose is to accept.

Constant

Procedures $\mathcal{C}_k^x$, where $x \in \{0,1\}$, check that a given address is a constant.

Start ID: The apple is at node $\mathbf{d}$ of level k, and $\mathbf{d}$ encodes a binary word w of multiexponential length $\exp_k(n)$. Local registers below node $\mathbf{d}$ are empty.

Claim: Procedure $\mathcal{C}_k^0$ (resp. $\mathcal{C}_k^1$) accepts iff the address of $\mathbf{d}$ is $\vec{0}$ (resp. $\vec{1}$).

Equality

Procedure $\mathcal{E}_k^l$, where $l > k$, verifies equality of two binary words.

Start ID: A start IDs of $\mathcal{E}_k^l$ has the apple at node $\mathbf{d}$, a root of a subtree of uniform depth l. (Then $\mathbf{d}$ is at level l.) At level k there is exactly one descendant $\mathbf{e}_O$ of $\mathbf{d}$ satisfying $T(\mathbf{e}_O)(\mathit{Old}) = 1$ and exactly one descendant $\mathbf{e}_N$ satisfying $T(\mathbf{e}_N)(\mathit{New}) = 1$. (There may be other nodes at level k as well, and it may happen that $\mathbf{e}_O = \mathbf{e}_N$.) All local registers below $\mathbf{e}_O$ and $\mathbf{e}_N$ are empty. Subtrees rooted at $\mathbf{e}_O$ and $\mathbf{e}_N$ encode binary words of length $\exp_k(n)$.

Claim: Procedure $\mathcal{E}_k^l$, initiated in a start ID, accepts iff the addresses of $\mathbf{e}_O$ and $\mathbf{e}_N$ are the same.

Successor

Binary words are identified with numbers so that the successor relation holds between strings of the form $w011\ldots1$ and $w100\ldots0$. Procedure $\mathcal{S}_k$ verifies this relation.

Start ID: The same as start ID of $\mathcal{E}_k^{k+1}$.

Claim: Procedure $\mathcal{S}_k$, initiated in a start ID, accepts iff the address of $\mathbf{e}_N$ is the successor of the address of $\mathbf{e}_O$.

Procedures

To provide a gentle introduction we begin our presentation with the relatively simple procedure $\mathcal{C}_k^0$; after that we proceed in the order of the previous subsection.

Procedure $\mathcal{C}_k^0$

We define our automata by mutual induction with respect to k. We begin with the relatively simple definition of $\mathcal{C}_k^0$, written in informal pseudo-code. For $k = 0$, the definition of $\mathcal{C}_k^0$ is a straightforward generalization of the code of $\mathcal{A}'$ in Example 8:

for $i = 1$ to n do [check L_i AND continue]; accept.

For $k > 0$, we assume that $\mathcal{C}^0_{k-1}$, $\mathcal{C}^1_{k-1}$, S_{k-1} have already been defined, and we construct $\mathcal{C}^0_k$, so that it executes the following algorithm. The almost identical definition of $\mathcal{C}^1_k$ is omitted.

1. Descend to a child; goto 2 AND goto 3;
2. Run $\mathcal{C}^0_{k-1}$ (accepting inside).
3. Check data register A_0; set register *Done*;
4. goto 5 OR goto 12;
5. Go up to $\mathbf{d}$;
6. Descend to a child;
7. goto 8 AND goto 3;
8. Set register *New*; go up to $\mathbf{d}$;
9. Descend to a child;
10. Check register *Done*; set register *Old*; go up to $\mathbf{d}$;
11. Run $\mathcal{S}_{k-1}$ (accepting inside);
12. Run $\mathcal{C}^1_{k-1}$ (accepting inside).

First, let us make an informal account of the way the procedure operates. When $\mathcal{C}^0_k$ is initiated in a start ID at a node $\mathbf{d}$ at level k, it attempts to verify that data register A_0 is set to 1 at every address. It begins with a child with address $\vec{0}$, guessing it non-deterministically. At this point the computation splits into two branches. One branch verifies the correctness of the guess by running $\mathcal{C}^0_{k-1}$ (and accepts if the verification is successful). Along the other branch we first check that A_0 is indeed set to 1, mark the present node as *Done*, and then proceed to another child of $\mathbf{d}$ (step 6). The main loop in steps 3–7 should now be taken for every address in the increasing order. Each time the body of the loop is executed, the machine verifies that the address of the current apple is a successor of another address which has already been processed. This is done with help of another universal split in step 7. A separate branch of computation is activated. Within that branch, the present node $\mathbf{e}$ is marked as *New*, then another child $\mathbf{e}'$ of $\mathbf{d}$ is selected and marked as *Old*. But first we check register *Done* at node $\mathbf{e}'$ to make sure that $\mathbf{e}'$ has been processed.[2] It remains to run $\mathcal{S}_{k-1}$ from node $\mathbf{d}$ to complete the verification branch (steps 8–11).

The main loop continues until we non-deterministically guess that we reached a node with address $\vec{1}$. This is verified by initiating $\mathcal{C}^1_{k-1}$, and then the procedure accepts.

Let us remark here that, although the above description of the algorithm is informal, it is precise enough to be implemented as an actual automaton, using a number of internal states proportional to n. Now we can show that $\mathcal{C}^1_k$ satisfies the specification.

($\Leftarrow$) Observe that in case the address of $\mathbf{d}$ encodes the word $w = \vec{0}$ and $\mathcal{C}^0_k$ is run from a correct start ID then the procedure may choose to take the child of $\mathbf{d}$ with address $\vec{0}$ in step 1 so that $\mathcal{C}^0_{k-1}$ accepts in step 2. Then all other children are chosen in step 6 in order of increasing addresses, so that it is always possible in step 9 to choose an appropriate predecessor address, guaranteeing termination in step 11. A more formal proof should go by induction with respect to the number of children of $\mathbf{d}$ marked as *Done*. Note that local registers at levels $k-2$ and below are empty and can be safely used by each procedure. Every branch of computation uses its own private copy of these registers. This way alternation helps to avoid the limitations of our non-erasable memory.

($\Rightarrow$) Suppose now that $\mathcal{C}^0_k$ accepts. Let l be the number of times the procedure enters step 4 in the accepting computation. Let D_i be the set of children of $\mathbf{d}$ marked as *Done* at

[2] It may happen that $\mathbf{e}' = \mathbf{e}$ but in this case the successor test will fail.

the i-th entry to step 4. Let a_i be the maximal address encoded by an element of D_i. By induction with respect to $l - i$ we show the following statement

> *For each accepting computation subtree of $\mathcal{C}^0_k$ started at the i-th entry to step 4 and for each address b such that $a_i < b < \exp_k(n)$, the node $\mathbf{d}$ has a child that encodes the number b and has A_0 set to* 1.

Indeed, for $i = l$, the set of addresses a such that $a_i < b < \exp_k(n)$ is empty, so the conclusion follows. If $i < l$ then an accepting computation must enter the loop and mark one child of $\mathbf{d}$ with *Done* and then come back to the step 4. We have two subcases here depending on the relation between the elements a_i and a_{i+1}. In case $a_i = a_{i+1}$ we observe that no node of the tree of knowledge could change in this turn of the loop (*Done* is only overwritten with the same value) so the conclusion follows by the induction hypothesis. In case $a_i \neq a_{i+1}$, there is $\mathbf{b}_i \in D_{i+1} - D_i$. Let b_i be the number encoded by $\mathbf{b}_i$. In steps 8–11 it is verified that $b_i = a + 1$, for some a encoded by $\mathbf{a} \in D_i$, but actually b must be a_i as otherwise $a_i = a_{i+1}$. This also means that $b_i = a_{i+1}$. Node $\mathbf{b}_i$ has A_0 set to 1, as this is verified in step 3. Since all other elements a such that $a_i < a < \exp_k(n)$ must satisfy $a_{i+1} < a < \exp_k(n)$, we obtain the conclusion by the induction hypothesis.

Now observe that at the first entry to step 4 only one child of $\mathbf{d}$ is marked as *Done*, and it must encode the address $\vec{0}$ (steps 1–2) with A_0 set (step 3). As the further computation accepts, we can apply the statement proven above for $i = 1$ and obtain that $\mathbf{d}$ has children that encode addresses b such that $0 < b < \exp_k(n)$ and all have A_0 set to 1. This applies also for the address 0. Since by assumption $\mathbf{d}$ encodes a a word of length $\exp_k(n)$, this must be the number of children of $\mathbf{d}$. Therefore $\mathbf{d}$ encodes $\vec{0}$.

Let us remark here that in step 6 the apple may be passed to a child already marked as *Done*, so that the main loop in steps 3–7 may be executed more times then needed and we effectively care about this case in the inductive step of the argument above.
A diggression before we proceed to the next procedure: The above algorithm can easily be adapted to verify if the binary string encoded by $\mathbf{d}$ belongs to any fixed regular language.

Procedure $\mathcal{M}_k$

We can now turn to the more complicated procedure $\mathcal{M}_k$. For the base case $k = 0$ we generalize the automaton $\mathcal{A}$ of Example 8:

$$\mathsf{for}\ i = 1\ \mathsf{to}\ n\ \mathsf{do}\ [\mathtt{set}\ L_i\ \mathsf{OR}\ \mathtt{set}\ R_i].$$

In the induction step we assume that procedures $\mathcal{M}_{k-1}$, $\mathcal{E}^l_{k-1}$, $\mathcal{S}_{k-1}$, $\mathcal{C}^0_{k-1}$, and $\mathcal{C}^1_{k-1}$, have already been defined, and we describe $\mathcal{M}_k$ as a pseudo-code "program" consisting of two phases. Recall that the computation begins at the root $\mathbf{d}$ of the word to be constructed.

Phase 1: At first, procedure $\mathcal{M}_k$ runs $\mathcal{M}_{k-1}$ in a loop. The number of iterations is chosen non-deterministically, but it is bounded due to the n-restrictedness condition, as each iteration creates a new child.

1. Create a new child and descend there;
2. Run $\mathcal{M}_{k-1}$;
3. Set register A_0 OR set register A_1;
4. go up; goto 1 (continue) OR goto 5 (enter Phase 2).

Note a subtlety: once a new child is created the computation must commence from a fixed initial state (for the appropriate depth). Our construction respects this restriction: we perform exactly the same actions for every new child.

An immediate inductive argument (for the loop in steps 1–4) shows that

1. The computation does not use (jumps, writes to or reads from registers at) any proper ancestor of **d**.
2. At the entry to step 4 the apple is back at **d**, and **d** has a non-empty set C of children with $|C| \leq \exp_k(n)$. Each element of C has either A_0 or A_1 set to 1 and starts a subtree that encodes a number in $\{0, \ldots, \exp_k(n) - 1\}$.

The inequality $|C| \leq \exp_k(n)$ is precisely the result of our n-restrictedness condition.

Phase 2: The second phase starts with the apple at node **d** and goes as follows:

5. Descend to a child; goto 6 (verify) AND goto 7 (continue);
6. Run $\mathcal{C}^0_{k-1}$ (accepting inside).
7. Set register *Steady*;
8. goto 9 OR goto 16;
9. Go up to **d**;
10. Descend to a child;
11. goto 12 (verify) AND goto 7 (continue);
12. Set register *New*; go up to **d**;
13. Descend to a child;
14. Check register *Steady*; set register *Old*; go up to **d**;
15. Run $\mathcal{S}_{k-1}$ (accepting inside);
16. Run $\mathcal{C}^1_{k-1}$ (verify) AND goto 17 (continue);
17. Go up to **d** (end state).

The second phase works very much like the procedure $\mathcal{C}^0_k$. In step 5 the computation splits into two branches. One proceeds (fingers crossed) along the main computation branch beginning at step 7. The other branch verifies that the present address is $\vec{0}$ and accepts. The whole computation can therefore accept only if the verification in step 6 was successful. In addition the auxiliary branch uses its own "private copy" of all resources, in particular it can set registers which remain empty for the main computation. Similar universal splits occur in steps 11 and 16. Note that registers *Old* and *New* remain intact outside of the subroutine 12–15. At the completion of the above we are back at node **d**. Again an immediate inductive argument (for the loop in steps 7–11) shows that:

1. The computation does not use (jumps to, writes to or reads from registers at) any proper ancestor of **d**.
2. Each time the computation reaches step 8, the apple is in a child of **d**, and **d** has a non-empty set C of children with $|C| \leq \exp_k(n)$. The set of numbers encoded by nodes in C is closed with respect to predecessor (in particular it contains zero).

Phase 2 reaches the end state only when it can verify that address $\vec{1}$ of length $\exp_k(n)$ is encoded by a child of **d** that is marked with *Steady*. With $\vec{1}$ marked as *Steady* and the closure with respect to predecessor we obtain that all addresses of length $\exp_k(n)$ must be encoded by children of **d**. And each of them only once, because the computation is n-restricted. This is exactly part 2 of the induction hypothesis for $\mathcal{M}_k$. Part 1 follows from (1) above.

▶ **Remark 10.** *Observe that this procedure may be easily adapted to serve as a non-deterministic generator of words of length* $\exp_k(n)$ *over arbitrary alphabet* Σ*. It is enough to use more registers and to adjust step 3 of the automaton* $\mathcal{M}_k$ *so that it chooses one of the registers corresponding to elements of* Σ *instead of* A_0 *or* A_1*.*

Procedure $\mathcal{S}_k$

Recall that we begin in a node **d** which has (among others) exactly one child marked as *Old* (i.e., satisfying $Old = 1$) and exactly one marked as *New*.

Subtrees rooted at these nodes are assumed to encode binary words w_{old} and w_{new} of length $\exp_k(n)$. We want to verify that $w_{old} = w011\ldots1$ and $w_{new} = w100\ldots0$, for some w. For $k = 0$ this can be done with a simple for loop. For $k > 0$, we process children of *Old* in order of increasing addresses. At each step we compare the data bit at the present node with the data bit at a child of *New* with the same address. The compared bits should match in phase 1 (we begin with more significant ones) until we non-deterministically discover the point where they begin to differ (phase 2).

We now describe $\mathcal{S}_k$ with a little more detail, but on a higher level of abstraction than the previous procedures. We believe that this account is still precise enough, and at the same time easier to understand. To make it even more comprehensive, let us first explain some of the phrases used below. For instance, "to descend to a child of *Old*" (step 1) means to descend to a child **e** of **d**, check **e**(*Old*), and then go to a child of **e**. The phrase "Universally verify that…" is understood as "Verify that… AND continue". (A similar construction was already used in the definitions of $\mathcal{C}^0_k$ and $\mathcal{M}_k$.) In step 2 this is equivalent to the statement "Run $\mathcal{C}^0_{k-1}$ AND goto 3". Similarly, in steps 7 and 13 the verification branch calls procedure $\mathcal{S}_{k-1}$, and steps 4, 9, 14 activate procedure $\mathcal{E}^{k+1}_{k-1}$.

1. Descend to a child of *Old*;
2. Universally verify that the present address consists of only zeros;
3. goto 4 (phase 1) OR goto 9 (end of phase 1);
4. Universally verify that the data bit at the present node is the same as the data bit of a child of *New* of the same address;
5. Mark the present node as *Done*; go up (to the node marked as *Old*);
6. Descend to a child;
7. Using S_{k-1}, universally verify that the present address is the successor of an address of a brother node already marked as *Done*;
8. goto 3;
9. Universally verify that the data bit at the present node is 0, while the data bit of a child of *New* of the same address is 1;
10. Mark the present node as *Gone*;
11. goto 12 (phase 2) OR goto 16 (end);
12. Go back to **d**; descend to a child of *Old*;
13. Universally verify that the present address is the successor of an address of a brother node already marked as *Gone*;
14. Universally verify that the data bit at the present node is 1, while the data bit of a child of *New* of the same address is 0;
15. Mark the present node as *Gone*; goto 11;
16. Run $\mathcal{C}^1_{k-1}$ (accepting inside).

Assuming that the start ID of $\mathcal{S}_k$ is as expected, we can now refer to the induction hypothesis about $\mathcal{C}^0_{k-1}$, $\mathcal{S}_{k-1}$, and $\mathcal{E}^l_{k-1}$. Indeed, all these procedures are run from their respective start IDs. In particular, local registers below level $k-1$ are available for use in the appropriate branches of computation. It follows that a successful computation of S_k is only possible when the successor relation indeed holds as required.

Procedure $\mathcal{E}_k^l$

This procedure works in a similar way as $\mathcal{S}_k$ except that only one phase is needed and the distance from $\mathbf{d}$ to *Old* and *New* may be larger. We skip the details, but we want to remark on one difference between $\mathcal{E}_k^l$ and $\mathcal{S}_k$. It may happen that either of these procedures is run from an ID where the same node of the tree is marked *Old* and *New*. This is not an obstacle: procedure $\mathcal{E}_k^l$ will accept in this case while $\mathcal{S}_k$ will not.

4.2 Simulation of a Turing Machine

The techniques introduced in Section 4.1 can be used to simulate a Turing Machine. Consider a deterministic Turing Machine $\mathcal{T}$ working in time $\exp_k(n^{\mathcal{O}(1)})$ and fix an input word $\mathbf{x}$ of length n. Without loss of generality[3] we can assume that the machine works exactly in time $\sqrt{\exp_k(n)} - 1$. Let $\Sigma = \Sigma_0 \cup (\Sigma_0 \times \Delta)$ where Σ_0 is the tape alphabet and Δ is the set of states of $\mathcal{T}$. We already know (see Remark 9) how to encode words over Σ using trees of knowledge.

We use a triple $\langle t, a, \mathtt{s} \rangle$ to express that the contents of the tape cell a at time t is $\mathtt{s}$. Here, $\mathtt{s} \in \Sigma$ is either a tape symbol of $\mathcal{T}$ or a tape symbol plus an internal state (in case $\mathcal{T}$ at time t is at position a). A computation of $\mathcal{T}$ is represented by a unique set of triples with only one $\langle t, a, \mathtt{s} \rangle$ for every t, a. Note that $a, t \in \{0, \ldots, \sqrt{exp_k(n)} - 1\}$. Consequently there are exactly $exp_k(n)$ pairs $\langle t, a \rangle$ and they can be identified with numbers less than $exp_k(n)$. The whole computation of machine $\mathcal{T}$ may therefore be seen as a word over Σ of length $exp_k(n)$. This word may now be encoded, as in Section 4.1, by a tree of knowledge of depth k and an appropriate dimension (extra data registers are needed to account for all elements of Σ). In this way we can represent a computation of $\mathcal{T}$ in the memory of an Eden automaton.

A slight adjustment of the automaton $\mathcal{M}_k$ of Section 4.1 (in step 3) yields a procedure to generate an arbitrary word over Σ of length $\exp_k(n)$.

Procedure $\mathcal{N}_k$

The definition of $\mathcal{N}_k$ is similar to that of $\mathcal{M}_k$, but now we have to only generate words representing accepting computations of $\mathcal{T}$. Therefore, $\mathcal{N}_k$ works in the following two phases:

1. It generates a sequence of triples.
2. It verifies that the set of triples represents a computation of $\mathcal{T}$.

Phase 1 is similar to phase 1 of $\mathcal{M}_k$ (see Remark 10). Phase 2 is more complicated, but it can similarly be related to phase 2 of $\mathcal{M}_k$. Steps 12–15 should be replaced with a longer verification routine. There are two subgoals of the routine:

1. To verify that the generated sequence of triples contains an encoding of the input word $\mathbf{x}$.
2. To verify that the sequence obeys the transition relation of $\mathcal{T}$.

For part (1) it has to be established that in every triple of the form $\langle 0, a, \mathtt{s} \rangle$, the value $\mathtt{s}$ is the symbol at position a in the initial configuration. To this end we use $n + 1$ new procedures $\mathcal{C}^a$, defined for $a \leq n$. Procedure $\mathcal{C}^a$ accepts from $\langle t, a', \mathtt{s} \rangle$ if $t = 0$, and $\mathtt{s} = x_a$, where $a = \min\{a', n\}$, and x_a is the appropriate symbol of the input (or blank for $a = n$).

[3] Using a routine padding technique one shows that every language in $\textsc{Dtime}(\exp_k(n^{\mathcal{O}(1)})$ reduces in polynomial time to one of time complexity $\exp_k(n-1)$, which is (for $k \geq 3$) less than the square root of $\exp_k(n)$.

The definition of $\mathcal{C}^a$ is similar to that of $\mathcal{C}^x_k$. Observe that procedures $\mathcal{C}^a$ are initiated in separated branches of computation so they can use the same registers.

We handle (2) by an iteration over triples $\langle\, t, a, \mathtt{s} \,\rangle$ for $t = 0, \ldots, \sqrt{\exp_k(n)} - 1$. The automaton expects that subtrees encoding triples $\langle\, t{-}1, a{-}1, \mathtt{s}_1 \,\rangle$, $\langle\, t{-}1, a, \mathtt{s}_2 \,\rangle$, $\langle\, t{-}1, a{+}1, \mathtt{s}_3 \,\rangle$, are also present. Those can be nondeterministically guessed and their roots appropriately marked (using four special registers for this purpose). Then we can run a subroutine $\mathcal{E}^{\mathtt{q}}$ (where $\mathtt{q}$ is a transition of $\mathcal{T}$) to confirm the guess. (The number of such subroutines is proportional to the size of the machine $\mathcal{T}$.) The definition of $\mathcal{E}^{\mathtt{q}}$ combines the tricks used in the construction of $\mathcal{S}_{k-1}$ and $\mathcal{E}^k_{k-1}$. An additional complication is that it must compare halves of words rather than the whole words (recall that we merge t and a in $\langle\, t, a, \mathtt{s} \,\rangle$ into a single word). This is not a real problem, as the end of the first half is identified by an address of the form $011\ldots1$. The construction of $\mathcal{E}^{\mathtt{q}}$, again, can be accomplished by a number of additional registers depending only on $\mathcal{T}$.

Automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$

The automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ first runs the procedure $\mathcal{N}_k$. Upon reaching the end state of $\mathcal{N}_k$ it checks that there is a triple $\langle\, t, a, \mathtt{s} \,\rangle$ where $\mathtt{s} = \langle\, \mathbf{a}, f \,\rangle$ and f is an accepting state of $\mathcal{T}$.

▶ **Lemma 11.** *Let $\mathcal{T}$ be a deterministic Turing Machine that works in time $\sqrt{\exp_k(n)} - 1$, and let $\boldsymbol{x}$ be a word of length n. The automaton $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ has an n-restricted accepting computation iff the machine $\mathcal{T}$ accepts $\boldsymbol{x}$.*

Proof. Suppose $\mathcal{T}$ accepts. By construction, the automaton $\mathcal{N}_k$ has a computation that reaches an end ID which properly encodes the computation of $\mathcal{T}$. All that remains is to verify that this computation contains an accepting ID. This amounts to a single non-deterministic `check`.

Now suppose that $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ has an accepting computation tree. This computation contains an end ID of $\mathcal{N}_k$, where the computation of $\mathcal{T}$ is properly encoded. Now there is no other way in which $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ can accept from such ID but to find an accepting state. So if $\mathcal{A}_{\mathcal{T},\mathbf{x}}$ is accepting, it must be the case that $\mathcal{T}$ accepts the word $\mathbf{x}$. ◀

The above combined with Lemma 7 yields a polynomial-time reduction of any language in $\textsc{Dtime}(\exp_k(n^{\mathcal{O}(1)})$ to Problem 3. We can thus conclude with the following theorem:

▶ **Theorem 12.** *The restricted decision problem for positive quantification is not elementary.*

We note that the above applies to monadic formulas (those involving only unary predicates). Indeed, the encoding in Section 3.2 did not require predicates of any higher arity.

5 Conclusion

We have demonstrated that the provability problem for intuitionistic logic with positive quantification becomes non-elementary under an apparently small restriction on proofs (computations). Technically, the only use of this restriction is in the definition of procedures $\mathcal{M}_k$ that generate representation for long strings of bits. Therefore, if an unrestricted implementation of $\mathcal{M}_k$ is possible then the original (unrestricted) problem is also not elementary.

The restriction we propose is a bound on a particular kind of a certain non-reusable resource. Under this restriction, the decidability of our formulas becomes immediate, as it reduces the search space to a finite size. In fact, the argument in e.g., the work by Dowek

and Jiang [6] or in the work by Mints [13] shows that the actual use of this resource in a proof of a formula φ is essentially equivalent to that in an $\mathcal{O}(n)$-restricted proof, where n is the size of φ. Still, the proof itself does not have to be n-restricted. Shall we prove it has, the general result will follow from our consideration. Although the opposite seems unlikely, the conjecture remains an open question.

References

1 Ulrich Berger, Kenji Miyamoto, Helmut Schwichtenberg, and Monika Seisenberger. Minlog: a tool for program extraction supporting algebras and coalgebras. In *Proc. of CALCO'11*, volume 6859 of *LNCS*, pages 393–399. Springer, 2011.

2 Egon Börger, Erich Grädel, and Yuri Gurevich. *The Classical Decision Problem*. Perspectives in Mathematical Logic. Springer, 1997.

3 Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda – a functional language with dependent types. In Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *LNCS*, pages 73–78. Springer, 2009.

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

5 The Coq Development Team. *The Coq Proof Assistant. Reference Manual*. INRIA, December 2011.

6 Gilles Dowek and Ying Jiang. Eigenvariables, bracketing and the decidability of positive minimal predicate logic. *Theoretical Computer Science*, 360(1–3):193–208, 2006.

7 Gilles Dowek and Ying Jiang. Enumerating proofs of positive formulae. *Computer Journal*, 52(7):799–807, 2009.

8 Georges Gonthier. The four colour theorem: Engineering of a formal proof. In D. Kapur, editor, *Computer Mathematics*, volume 5081 of *LNCS*. Springer, 2008.

9 Georges Gonthier. Advances in the formalization of the odd order theorem. In M. van Eekelen, H. Geuvers, J. Schmaltz, and F. Wiedijk, editors, *Interactive Theorem Proving*, volume 6898 of *LNCS*, page 2. Springer, 2011.

10 Gerwin Klein et al. seL4: Formal verification of an OS kernel. *Communications of the ACM*, 53(6):107–115, 2010.

11 Daniel Leivant. Monotonic use of space and computational complexity over abstract structures. Technical Report CMU-CS-89-21, Carnegie Mellon University, 1989.

12 Xavier Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

13 Grigori E. Mints. Solvability of the problem of deducibility in LJ for a class of formulas not containing negative occurrences of quantifiers. *Steklov Inst.*, 98:135–145, 1968.

14 Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL – A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, 2002.

15 Jens Otten. ileanTAP: An intuitionistic theorem prover. In D. Galmiche, editor, *Proc. of TABLEAUX'97*, volume 1227 of *LNCS*, pages 307–312. Springer, 1997.

16 Ivar Rummelhoff. *Polymorphic* Π1 *Types and a Simple Approach to Propositions, Types and Sets*. PhD thesis, University of Oslo, 2007.

17 Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. Positive logic is not elementary. Presentation at the Highlights conference, 2013.

18 Aleksy Schubert, Paweł Urzyczyn, and Daria Walukiewicz-Chrząszcz. How hard is positive quantification? In preparation, 2014.

19 Aleksy Schubert, Paweł Urzyczyn, and Konrad Zdanowski. On the Mints hierarchy in first-order intuitionistic logic. In A. Pitts, editor, *Foundations of Software Science and Computation Structures 2015*, volume 9034 of *LNCS*, pages 451–465. Springer, 2015.

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

21 Hao Wang. A variant to Turing's theory of computing machines. *Journal of the ACM*, 4(1):63–92, 1957.

22 Tao Xue and Qichao Xuan. Proof search and counter model of positive minimal predicate logic. *ENTCS*, 212(0):87–102, 2008.

On Isomorphism of Dependent Products in a Typed Logical Framework*

Sergei Soloviev[1,2]

1 **IRIT, University of Toulouse**
118 route de Narbonne, 31062 Toulouse, France
soloviev@irit.fr
2 **associated researcher at**
ITMO University
St. Petersburg, Russia

Abstract

A complete decision procedure for isomorphism of kinds that contain only dependent product, constant *Type* and variables is obtained. All proofs are done using Z. Luo's typed logical framework. They can be easily transferred to a large class of type theories with dependent product.

1998 ACM Subject Classification F.4.1 Lambda calculus and related systems, D.2.13 Reuse models, D.3.3 Frameworks

Keywords and phrases Isomorphism of types, dependent product, logical framework

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.274

1 Introduction

Why an axiomatization of the isomorphism relation between types in dependent type systems (based on type rewriting, as in the case of simply-typed λ-calculus or system F, see, *e.g.*, [1, 3, 15]) was never considered? Why no complete decision procedure for this relation was developed there? Isomorphism of dependent types is used to some extent in proof assistants based on dependent type systems, such as Coq (*cf.* [4]). We could find only one paper by D. Delahaye [5] where the author tries to explore type isomorphisms in the Calculus of Constructions along the lines used in the above-mentioned papers. On a theoretical side, isomorphisms play also an important role in the study of Univalent Foundations [9]. There are some studies of isomorphisms of inductive types [2, 6], but little is done on isomorphisms even in the "core" of logical frameworks (including, *e.g.*, dependent product).

In the paper [5] dependent product *and* dependent sum are considered but no complete axiomatisation (suitable for "non-contextual" rewriting) or complete decision procedure is obtained. As Delahaye writes:

- we have developed a theory Th^{ECCE} with "ad hoc" contextual rules, which is sound for $ECCE$;
- we have made contextual restrictions on Th^{ECCE} to build a decision procedure Dec^{Coq} which is sound for Th^{ECCE} and which is an approximation of the contextual part of Th^{ECCE};
- we have implemented Dec^{Coq} in a tool called $SearchIsos$.

* This work was partially supported by Government of the Russian Federation Grant 074-U01.

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 274–287

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

It should be said that some points concerning the notion of isomorphism in *ECCE* used in [5] remain not very clear. In particular, he needs η-rules to justify all axioms he considers. It is known that with Σ-types and cumulative hierarchy, the CR-property does not hold in the presence of η-contraction scheme (and similar scheme for surjective pairing) [11], p.50.

In this paper we are going to obtain a *complete* decision procedure in a "core" system that contains only dependent product, constant $Type$ and variables. We shall consider mostly the logical framework proposed by Z. Luo [11]. It is sufficiently close to the Martin-Löf logical framework or to the Calculus of Constructions, but in difference from Martin-Löf's original system it is typed, and in difference from the Calculus of Constructions it has the explicit equality rules even for standard $\beta\eta$-conversions, and this is more convenient for the study that concerns both isomorphism and equality. In difference from Edinburgh LF it permits to specify other type theories. This system (without Σ-types and cumulative hierarchy) is confluent with respect to $\beta\eta$-reductions.

We shall use the notation from [11]. In particular, $(x : K)K'$ will denote dependent product and $[x : K]P$ abstraction (instead of frequently used $\Pi x : K.K'$ and $\lambda x : K.P$). If $x : K$ does not occur freely in K', it will be written $(K)K'$ (that corresponds to $K \to K'$ in simply typed lambda calculus).

As to the above mentioned "core part", an answer may be that the isomorphism relation between dependent product kinds seems at a first glance too limited to be interesting. Among "basic" isomorphisms, there is only one obvious isomorphism that corresponds to the isomorphism $A \to (B \to C) \sim B \to (A \to C)$ of simply-typed λ-calculus. The corresponding isomorphism in dependent type case is

$$\Gamma \vdash (x : A)(y : B)C \sim (y : B)(x : A)C.$$

Here A, B, C are kinds, $(x : A)D$ means dependent product. In difference from simply-typed calculus we need a context Γ because A, B, C may contain free variables. The variable x must be not free in B and y in A, and $(x : A)(y : B)C$, $(y : B)(x : A)C$ must be well formed kinds in Γ.

Notice that *a priori* it does not exclude the existence of isomorphisms that are *not* generated by this basic isomorphism (*cf.* [7]).

In fact, though, there are some other aspects that make even the isomorphisms in the "core part" of dependent type systems interesting. The role of contexts (variable declarations) is to be taken into account. The equality of kinds is non-trivial and it has an influence on (the defintion of) the isomorphisms: for example, the condition that x is not free in B above may be not satisfied but B may be equal to B_0 that does not contain x free.

In the "core part" itself the role of contexts is rather superficial, but it shows what is to be expected if we consider more sophisticated type theories defined using logical frameworks.

The next aspect is more important. It is illustrated by the following example. Let $\Gamma \vdash A \sim A'$. Consider $\Gamma' \vdash (x : A)B$. Let $\Gamma \vdash P : (x : A)A'$ be the term that represents the isomorphism between A and A' and $\Gamma \vdash P' : (x : A')A$ the term that represents its inverse (in this case x is not free in A' and x' is not free in A). Then, in difference from the simply-typed case where $(x : A)B \sim (x : A')B$, the isomorphism P' appears inside B:

$$\Gamma \vdash (x : A)B \sim (x' : A')[(P'x')/x]B$$

($[(P'x')/x]$ denotes substitution). Notice that there may be many mutually inverse isomorphisms between A and A' (represented by $P_1, P'_1, \ldots, P_n, P'_n, \ldots$ and the structure of the "target" type $(x : A')[(P'_i x)/x]B$ depends on their choice. (This was noticed already in [5].) Thus, if we see the isomorphic transformation as rewriting, this rewriting is not *local*, and there is

little hope that one can describe the isomorphism relation between types using rewriting rules for types[1] as in, *e.g.*, [1, 3, 15].

2 Basic definitions

We consider Z. Luo's typed logical framework LF [11].

Because LF is mostly used to specify type theories, types in LF are called kinds (to distinguish them from types in the specified type theories). In LF there are five forms of judgements (below $\Gamma \vdash J$ will be sometimes used as a generic notation for one of these five judgement forms):

- $\Gamma \vdash \mathbf{valid}$ (Γ is a valid context);
- $\Gamma \vdash K\mathbf{kind}$ (K is a kind in the context Γ);
- $\Gamma \vdash k : K$ (k is an object of the kind K);
- $\Gamma \vdash k = k' : K$ (k and k' are equal objects of the kind K);
- $\Gamma \vdash K = K'$ (K and K' are equal kinds in Γ).

There are the following inference rules in LF (we use here an equivalent formulation which is more convenient proof-theoretically, *cf.* [14]):

Contexts and assumptions

$$(1.1)\ \frac{}{<>\vdash \mathbf{valid}} \qquad (1.2)\ \frac{\Gamma \vdash K\mathbf{kind}\ \ x \notin FV(\Gamma)}{\Gamma, x:\ K \vdash \mathbf{valid}} \qquad (1.3)\frac{\Gamma, x:\ K, \Gamma' \vdash \mathbf{valid}}{\Gamma, x:\ K, \Gamma' \vdash x:\ K}$$

$$\frac{\Gamma_1, \Gamma_2 \vdash J\ \ \Gamma_1, \Gamma_3 \vdash \mathbf{valid}}{\Gamma_1, \Gamma_3, \Gamma_2 \vdash J}(wkn)$$

$$(\text{where } FV(\Gamma_2) \cap FV(\Gamma_3) = \emptyset).$$

General equality rules

$$(2.1)\ \frac{\Gamma \vdash K\mathbf{kind}}{\Gamma \vdash K = K} \qquad (2.2)\ \frac{\Gamma \vdash K = K'}{\Gamma \vdash K' = K} \qquad (2.3)\ \frac{\Gamma \vdash K = K'\ \ \Gamma \vdash K' = K''}{\Gamma \vdash K = K''}$$

$$(2.4)\ \frac{\Gamma \vdash k : K}{\Gamma \vdash k = k : K} \qquad (2.5)\ \frac{\Gamma \vdash k = k' : K}{\Gamma \vdash k' = k :\ K} \qquad (2.6)\frac{\Gamma \vdash k = k' : K\ \ \Gamma \vdash k' = k'' : K}{\Gamma \vdash k = k'' :\ K}$$

Retyping rules

$$(3.1)\frac{\Gamma \vdash k :\ K\ \ \Gamma \vdash K = K'}{\Gamma \vdash k :\ K'} \quad (3.2)\frac{\Gamma \vdash k = k' :\ K\ \ \Gamma \vdash K = K'}{\Gamma \vdash k = k' :\ K'}$$

$$(3.3)\frac{\Gamma, x : K, \Gamma' \vdash J\ \ \Gamma \vdash K = K'}{\Gamma, x : K', \Gamma' \vdash J}$$

[1] Maybe, it is better to say "non-contextual" instead of "local". But what is needed here is more than dependency on context of the applicability of a rewriting rule. In fact all occurrences of x (indefinitely many) must be simultaneously replaced by $P'x'$. The inclusion of an explicit substitution rule as a part of rewriting process may have its own drawbacks. The rewriting rules considered in [5] that take into account this observation are called there "contextual", but to us this terminology does not seem perfect. Indeed, the "context" has to be changed simultaneously, otherwise at some point the expression will not be well typed. There is also some confusion of the rewriting "context" in this sense, and the usual type-theoretical contexts of variable declarations.

The kind Type

$$(4.1)\frac{\Gamma \vdash \mathbf{valid}}{\Gamma \vdash Type\mathbf{kind}} \quad (4.2)\frac{\Gamma \vdash A : Type}{\Gamma \vdash El(A)\mathbf{kind}} \quad (4.3)\frac{\Gamma \vdash A = B : Type}{\Gamma \vdash El(A) = El(B)}$$

Dependent product (kinds and terms)[2]

$$(5.1)\frac{\Gamma, x : K \vdash K'\mathbf{kind}}{\Gamma \vdash (x : K)K'\mathbf{kind}} \qquad (5.2)\frac{\Gamma, x : K_1 \vdash K_1' = K_2' \quad \Gamma \vdash K_1 = K_2}{\Gamma \vdash (x : K_1)K_1' = (x : K_2)K_2'}$$

$$(5.3)\frac{\Gamma, x : K \vdash k : K'}{\Gamma \vdash [x : K]k : (x : K)K'} \qquad (5.4)\frac{\Gamma, x : K_1 \vdash k_1 = k_2 : K \quad \Gamma \vdash K_1 = K_2}{\Gamma \vdash [x : K_1]k_1 = [x : K_2]k_2 : (x : K_1)K}$$

$$(5.5)\frac{\Gamma \vdash f : (x : K)K' \quad \Gamma \vdash k : K}{\Gamma \vdash f(k) : [k/x]K'} \quad (5.6)\frac{\Gamma \vdash f = f' : (x : K)K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma \vdash f(k_1) = f'(k_2) : [k_1/x]K'}$$

$$(5.7)\frac{\Gamma, x : K \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma \vdash ([x : K]k')k = [k/x]k' : [k/x]K'} \quad (5.8)\frac{\Gamma \vdash f : (x : K)K' \quad x \notin FV(\Gamma)}{\Gamma \vdash [x : K]f(x) = f : (x : K)K'}$$

Substitution rules

$$(6.1)\frac{\Gamma, x : K, \Gamma' \vdash \mathbf{valid} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash \mathbf{valid}} \qquad (6.2)\frac{\Gamma, x : K, \Gamma' \vdash K'\mathbf{kind} \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K'\mathbf{kind}}$$

$$(6.3)\frac{\Gamma, x : K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' : [k/x]K'} \qquad (6.4)\ \frac{\Gamma, x : K, \Gamma' \vdash K' = K'' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]K' = [k/x]K''}$$

$$(6.5)\frac{\Gamma, x : K, \Gamma' \vdash k' = k'' : K' \quad \Gamma \vdash k : K}{\Gamma, [k/x]\Gamma' \vdash [k/x]k' = [k/x]k'' : K'}$$

$$(6.6)\ \frac{\Gamma, x : K, \Gamma' \vdash K'\mathbf{kind} \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma, [k_1/x]\Gamma' \vdash [k_1/x]K' = [k_2/x]K'}$$

$$(6.7)\ \frac{\Gamma, x : K, \Gamma' \vdash k' : K' \quad \Gamma \vdash k_1 = k_2 : K}{\Gamma, [k/x]\Gamma' \vdash [k_1/x]k' = [k_1/x]k' : [k/x]K'}$$

In the syntax of LF $(x : K)K'$ denotes dependent product, and $[x : K]k$ denotes abstraction, x is considered as bound in K' and k respectively. In case when x is not free in K' we shall write $(K)K'$ instead of $(x : K)K'$. We shall use $\equiv$ for syntactic identity.

One of the fundamental properties of derivations in LF is that the inferences of substitutions, *wkn* and context-retyping 3.3 that create problems with structural induction on derivations can be eliminated, *i.e.*, a judgement is derivable iff it has a substitution, context-retyping and *wkn*-free derivation ([14], Theorem 3.1, [12], Definition 3.12 and Algorithm 3.13).

In [12] such derivations are called canonical (Definition 3.12). The following technical lemmas are easily proved by induction on the size of canonical derivation in LF.

▶ **Lemma 1.** *Let $\Gamma, \Gamma', \Gamma'' \vdash J$ be any judgement derivable in LF. If the variables from Γ' do not occur into Γ'' and J, then $\Gamma, \Gamma'' \vdash J$ is derivable.*

[2] To facilitate reading, let us notice that the syntax of raw kinds and terms is very simple:
$K ::= Type \,|\, El(P) \,|\, (x : K)K', \quad P ::= x \,|(PQ)\,|[x : K]P.$

Since in LF types (*kinds*) of variables may depend on terms (other variables) the variables cannot any more be freely permuted. Let us formulate some statements (without detailed proofs) that we shall use below.

Let us consider the list of variables with *kinds*, $u_1 : Q_1, \ldots, u_k : Q_k$. Let $u_i \lhd u_j$ denote that u_i occurs in the kind Q_j of u_j. The same applies to prefixes like $(u_1 : Q_1) \ldots (u_k : Q_k)Q$ and $[u_1 : Q_1] \ldots [u_k : Q_k]Q$.

Let $u_1 : Q_1, \ldots, u_k : Q_k$ be part of a valid context (respectively $(u_1 : Q_1) \ldots (u_k : Q_k)Q$, $[u_1 : Q_1] \ldots [u_k : Q_k]Q$ be part of derivable *kind* or term). In this case the relation $\lhd$ generates a partial order on indexes $1, \ldots, k$ which we shall denote by $\lhd*$.

▶ **Lemma 2.** *Consider the judgements*

$$\Gamma, x_1 : K_1, \ldots, x_n : K_n, \Gamma' \vdash \mathbf{valid},$$

$$\Gamma \vdash (x_1 : K_1) \ldots (x_n : K_n)K_0\mathbf{kind},$$

$$\Gamma \vdash [x_1 : K_1] \ldots [x_n : K_n]P : (x_1 : K_1) \ldots (x_n : K_n)K_0$$

in LF. For any permutation σ that respects the order $\lhd*$,

$$\Gamma, x_{\sigma_1} : K_{\sigma_1}, \ldots, x_{\sigma_n} : K_{\sigma_n}, \Gamma' \vdash \mathbf{valid},$$

$$\Gamma \vdash (x_{\sigma_1} : K_{\sigma_1}) \ldots (x_{\sigma_n} : K_{\sigma_n})K_0\mathbf{kind},$$

$$\Gamma \vdash [x_{\sigma_1} : K_{\sigma_1}] \ldots [x_{\sigma_n} : K_{\sigma_n}]P : (x_{\sigma_1} : K_{\sigma_1}) \ldots (x_{\sigma_n} : K_{\sigma_n})K_0$$

are derivable in LF.

Besides standard equality rules, equality in LF is defined by the rules (5.7, 5.8). Obviously, it is based on β and η conversions (incorporated explicitly using 5.7 and 5.8). This permits to define conversions in a more familiar way.

▶ **Proposition 3.**

1. *Let J be an LF-judgement (of any of the five forms described above) and v an occurrence of an expression either of the form $([x : K]P)S$ or of the form $[x : K](Px)$ with x not free in P. Let J' be obtained by replacement of v by the occurrence of $[S/x]P$ or P respectively. Then J is derivable in LF iff J' is derivable. (We shall say that one is obtained from another by β, respectively η conversion.)*
2. *Let J be of the form $\Gamma \vdash K\mathbf{kind}$ or $\Gamma \vdash P : K$ respectively and v belong to K (respectively, to P). Let K', respectively, P' be obtained from K (k') as in 1. If J is derivable, the equality $\Gamma \vdash K = K'$, respectively, $\Gamma \vdash P = P' : K$ is derivable.*

Proof.

1. By induction on the length of a canonical derivation of J.
2. By induction on the length of a canonical derivation of J and (for one of implications) on the length of series of conversions. ◀

We shall use the fact that LF is strongly normalizing and has the Church-Rosser property with respect to β- and η-reductions, see H. Goguen's thesis [8], and also [13]. H. Goguen applied typed operational semantics to LF and its extension UTT to prove these results; L. Marie-Magdeleine in [13] applied Goguen's method to UTT with certain additional equality rules. We do not always need SN and CR in our proofs, but since we want to concentrate our attention on isomorphisms, the use of SN and CR permits to make some shortcuts.

In simply-typed λ-calculus the definition of invertibility of terms may use contexts (that include free term variables). This is relevant for the study of retractions [16]. However, type variables and terms variables are completely separated, and to describe the isomorphisms of types it is enough to consider closed terms [3]. Equality of types coincides with identity. One says that the types A, B are isomorphic iff there exist terms $P : A \to B$ and $P' : B \to A$ such that $P' \circ P =_{\beta\eta} \lambda x : A.x$ and $P \circ P' =_{\beta\eta} \lambda x : B.x$.

In dependent type systems the equality of types (kinds) depends on equality of terms. Some variables from the context may occur in kinds whose isomorphism we want to check. This motivates the following definition.

▶ **Definition 4.** Let $\Gamma \vdash K\mathbf{kind}$ and $\Gamma \vdash K'\mathbf{kind}$. We shall say that K, K' are isomorphic in Γ iff there exist terms $\Gamma \vdash P : (K)K'$, $\Gamma \vdash P' : (K')K$ such that

$$(*)\ \ \Gamma \vdash P' \circ P = [x : K]x : (K)K, \quad \Gamma \vdash P \circ P' = [x : K']x : (K')K'.$$

▶ Remark 5. Equal kinds may contain different free variables, and it has to be taken into account. If we consider $\beta\eta$-normal forms of K and K', they may not contain some free variables that are present in K and K'. The normal forms may be well-formed in a more narrow context Γ_0. Still, the isomorphism of K and K' will not hold in Γ_0 because Γ is necessary to prove the equality of kinds to their normal forms. In case of $\beta\eta$-equality one may try to define some sort of "minimal" context, but when the extensions of LF are more "exotic", this may be not possible (at the moment, we study one such extension, a generalization to dependent type systems of axiom C [10]).

3 Isomorphism of Kinds in LF

At a first glance, the theory of isomorphisms in LF cannot be very interesting. Indeed, with respect to the LF-equality there exists only one "basic" isomorphism, but as it turns out even this basic isomorphism generates in LF much more intricate isomorphism relation than in the simply-typed case.

▶ **Example 6.** Let $\Gamma \vdash (x : K_1)(y : K_2)K\mathbf{kind}$ be derivable in LF and $x \notin FV(K_2)$. Then $\Gamma \vdash (y : K_2)(x : K_1)K\mathbf{kind}$ is derivable and $(x : K_1)(y : K_2)K \sim (y : K_2)(x : K_1)K$ in Γ. The terms

$$[z : (x : K_1)(y : K_2)K][y : K_2][x : K_1](zxy),$$

$$[z : (y : K_2)(x : K_1)K][x : K_1][y : K_2](zyx)$$

are mutually inverse isomorphisms between these *kinds*.

This example corresponds directly to the well known example of isomorphism in simply-typed lambda calculus. In difference from simply-typed λ-calculus, there are some technical points that have to be proved, such as the fact that the derivability of $\Gamma \vdash (y : K_2)(x : K_1)K\mathbf{kind}$ follows from the derivability of $\Gamma \vdash (x : K_1)(y : K_2)K\mathbf{kind}$. For example, the derivability of $\Gamma \vdash (x : K_1)(y : K_2)K\mathbf{kind}$ implies the derivability of $\Gamma, x : K_1, y : K_2 \vdash K\mathbf{kind}$ and this implies the derivability of $\Gamma, y : K_2, x : K_1 \vdash K\mathbf{kind}$ because $x \notin FV(K_2)$ (cf. Lemma 2).

▶ **Example 7.** Let in the previous example $\Gamma \vdash K_1 = K_2$. Then there exists at least two isomorphisms between $(x : K_1)(y : K_2)K$ and $(y : K_2)(x : K_1)K$ in Γ. Indeed, one isomorphism is the identity isomorphism, and another is the isomorphism considered in the previous example.

The following example shows the "non-locality" of syntactic rewriting relation associated with isomorphisms in LF.

▶ **Example 8.** Let $\Gamma \vdash (x : K_1)K\mathbf{kind}$ be derivable in LF. Let $K_1 \sim K_2$ in Γ, and $\Gamma \vdash P : (K_1)K_2$, $\Gamma \vdash P' : (K_2)K_1$ be mutually inverse isomorphisms. Then $(x : K_1)K \sim (x : K_2)[(P'x)/x]K$ in Γ. The isomorphism from the first to the second kind is given by the following term:

$$\Gamma \vdash [z : (x : K_1)K][x : K_2](z(P'x)) : ((x : K_1)K)(x : K_2)[(P'x)/x]K,$$

and its inverse by

$$\Gamma \vdash [z : (x : K_2)[(P'x)/x]K][x : K_1](z(Px)) : ((x : K_2)[(P'x)/x]K)(x : K_1)K.$$

Notice that after second substitution (generated by the application of z in the second line) P' and P being mutually inverse isomorphisms will cancel each other. Notice also that since P and P' may be not a unique pair of isomorphisms between K_1 and K_2, the replacement of K_1 by K_2 does not uniquely determine the "target" kind. We cannot merely replace K_1 by K_2 (without introducing P' in K) because the correct kinding inside K may be lost.

Let $\Gamma \vdash P : El(A)$ be provable in LF. Then P is either a variable or an application. Formal proof can be done by induction on the length of derivation of $\Gamma \vdash P : El(A)$.

▶ **Lemma 9.** *Let* $\Gamma \vdash El(A)\mathbf{kind}$ *and* $\Gamma \vdash El(B)\mathbf{kind}$. *Then* $\Gamma \vdash El(A) \sim El(B)$ *iff* $\Gamma \vdash El(A) = El(B)$. *The isomorphism between* $El(A)$ *and* $El(B)$ *is unique up to equality in LF and is represented by the term* $\Gamma \vdash [x : El(A)]x : (El(A))El(B)$.

Proof. Consider the non-trivial "if". Assume there exist mutually inverse isomorphisms $\Gamma \vdash P : (El(A))El(B)$ and $\Gamma \vdash P' : (El(B))El(A)$. That is, the compositions of P and P' are equal to identities:

$$\Gamma \vdash [x : El(A)](P'(Px)) = [x : El(A)]x : (El(A))El(A),$$

$$\Gamma \vdash [x : El(B)](P(P'x)) = [x : El(B)]x : (El(B))El(B)$$

(with x fresh).

Without loss of generality we may assume that each of P, P' is normal. Consider, e.g., P'. It may have either the form $[y : El(B)](zk_1 \dots k_n)$ or the form $zk_1 \dots k_n$ (z being a variable). It is easily seen that in the second case the whole cannot normalize to $[x : El(A)]x$. In the first case, if it normalizes to $[x : El(A)]x$, n must be 1 and $[(Px)/y]k_1$ must normalize to x. Similar analysis of the form of P leads to the conclusion of the lemma. ◀

▶ **Theorem 10.** *Let* $\Gamma \vdash K\mathbf{kind}$ *in LF. Then:*

1. *the number of kinds (considered up to equality) that are isomorphic to* K *in LF in the context* Γ *is finite;*
2. *for every kind* $\Gamma \vdash K'\mathbf{kind}$ *such that* $K \sim K'$ *in* Γ, *the number of isomorphisms between* K *and* K' *in* Γ *is finite;*
3. *there exists an algorithm that lists all these isomorphisms (and kinds).*

First, let us notice that if $\Gamma \vdash K\mathbf{kind}$ is derivable in LF then K has either the form $(x_1 : K_1) \dots (x_n : K_n)El(A)$ or the form $(x_1 : K_1) \dots (x_n : K_n)\mathbf{Type}$. This can be easily proved by induction on the derivation of $\Gamma \vdash K\mathbf{kind}$ in LF.

Proof of Theorem 10. The proof will proceed by induction on rank of K which is defined as follows.

▶ **Definition 11.** If $K \equiv \mathbf{Type}$ or $K \equiv El(A)$ then $rank(K) = 0$. If $K \equiv (x : K_1)K_2$ then $rank(K) = max(rank(K_1), rank(K_2)) + 1$.

▶ Remark 12. The $rank(K)$ is not changed by β- and η-reductions inside K and by substitution: $rank(K) = rank([k/x]K)$.

The *base case* of induction is assured by Lemma 9.

To proceed, we shall use type erasure and Dezani's theorem about invertible terms in untyped λ-calculus[3] as in [1, 3]. Of course, some modifications to take into account dependent types will be necessary. Before we continue with the proof of the theorem, several definitions and auxiliary statements are needed.

▶ **Definition 13.** Let $\Gamma \vdash P : K$ in LF. By $e(P)$ we shall denote the λ-term obtained by erasure of all kind-labels in P (and replacement of all expressions $[x]$ by λx). We shall call term variables of P all variables that occur in $e(P)$.

The following definition is a refined (equivalent) reformulation of definition 1.9.2 of [3].

▶ **Definition 14.** An untyped λ-term M with one free variable x is a finite hereditary permutation (f.h.p.) iff

- $M \equiv x$, or
- there exists a permutation $\sigma : n \to n$ such that $M \equiv \lambda x_{\sigma_1} \ldots .x_{\sigma_n}.xP_1 \ldots P_n$ (the only free variable of M is x, and its unique occurrence is explicitly shown) where the only free variable of P_i is x_i and P_i is a finite hereditary permutation (for all $1 \leq i \leq n$).
- If M is a f.h.p. then the term $\lambda x.M$ will be called its closure. We shall also say that it is closed finite hereditary permutation (c.f.h.p.). The notion of c.f.h.p. corresponds to f.h.p. of [3].

▶ Remark 15. In "standard" cases the passage from the term P such that $e(P)$ is a f.h.p. to the term whose erasure is a c.f.h.p. is done by a single abstraction:

$$\frac{\Gamma, z : K \vdash P : K'}{\Gamma \vdash [z : K]P : (z : K)K'}.$$

We do not "abstract" the "head variable" of a f.h.p. because sometimes we want to by-step the problem of permutability of variables in LF if the head variable is not the rightmost variable of a context.

The result similar to simply-typed λ-calculus holds.

▶ **Proposition 16** (cf. Theorem 1.9.5 of [3]). *If $\Gamma \vdash P : (K)K'$ and $\Gamma \vdash P' : (K')K$ are mutually inverse terms in LF then $e(P)$ and $e(P')$ are c.f.h.p.*

If $e(P)$ is a c.f.h.p. then P has the structure

$$[z : (x_1 : K_1) \ldots (x_n : K_n)K_0][x'_{\sigma_1} : K'_{\sigma_1}] \ldots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \ldots P_n).$$

When we consider invertible terms, we always can assume that they are normal and in such a form.

[3] Probably the "simply-typed erasure": to replace all occurrences of $El(A)$ by El (considered as another constant *kind*) will work as well, but it seems that a fully justified application of this method may need as much technical lemmas as the proof that we propose below.

In difference from simply-typed λ-calculus, there are additional constraints on the possible permutations in LF, because some of the types K_j may depend on variables $x_i, i < j$, and in this case permutation of x_i and x_j destroys typability. As a consequence, if $e(P)$ is a f.h.p. then the original term is not necessarily well typed.

▶ **Example 17.** The term

$$P \equiv [z : (x : K_1)(y : K_2(x))K][y : K_2(x)][x : K_1](zxy)$$

is not well typed in LF, but $e(P)$ is a c.f.h.p.

Let us prove some lemmas concerning properties of well typed terms P such that $e(P)$ is a f.h.p. (they can be easily reformulated for c.f.h.p.)

▶ **Lemma 18.** *Let $\Gamma \vdash P : K'$ be derivable in LF, and $e(P)$ be a f.h.p. with head variable $z : K$, $K \equiv (x_1 : K_1) \dots (x_n : K_n)K_0$. Let*

$$P \equiv [x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n).$$

If x is a free variable that occurs in $P_1, \dots, P_n$ then it occurs in the kind of z.

Proof. There is no occurrence of x as term variable of f.h.p. because of the properties of f.h.p. Let there be an occurrence of x into *kinds* of variables in P_i. Notice that since P is well typed, $P_1 : K_1$ (in appropriate context), $P_2 : [P_1/x_1]K_2, \dots$, $P_n : [P_{n-1}/x_{n-1}](\dots[P_1/x_1]K_n$, $zP_1 \dots P_n : [P_n/x_n](\dots[P_1/x_1]K_0 = K'_0$.

Consider the Böhm-tree of $e(P)$ [1, 3]. We order the paths (from the root to nodes, not necessarily to leaves) lexicographically, in such a way that the path "more to the left" is less that the paths "more to the right". Now, we may find the smallest path such that the variable in some P_i that corresponds to the occurrence at its end contains x in its *kind*.

If the length of the path is 1, the occurrence lies in the prefix of P_i, and a matching occurrence of x into $[P_{i-1}/x_{i-1}](\dots[P_1/x_1]K_i$ must exist. Indeed, it cannot come from $P_j, j < i$ due to the choice of the path, so it comes from K_i.

Now, assume that the path is longer. Then we obtain a contradiction. Indeed, x must occur into the "abstracted" prefix of some subterm of some P_i, *i.e.*, it lies within an occurrence of the form $yQ_1 \dots Q_k$, and belongs to the prefix of some of Q_i. As above, we arrive at the conclusion that a matching occurrence into *kind* of y must exist, but y belongs to the abstracted prefix at the previous node of the same path. ◀

▶ **Corollary 19.** *Let $\Gamma \vdash P : K'$ be derivable in LF, $e(P)$ be a f.h.p. and z occur as the "head variable" of P. Then there is no other occurrences of z into P, even in the kinds of other variables.*

Proof. Since $e(P)$ is a f.h.p. z could occur (except the "head") only into kinds of variables in P. But then by the previous lemma it must occur into its own *kind* and this is impossible. ◀

▶ **Lemma 20.** *Let (as above) $\Gamma \vdash P : K'$ be derivable in LF, and $e(P)$ be a f.h.p. The free variable x occurs in K' iff it occurs in the kind of the head variable of P.*

Proof. As above, $P \equiv [x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n)$, where $z : (x_1 : K_1) \dots (x_n : K_n)K_0$. The variable x'_i is the head variable of P_i (by properties of f.h.p.).

We proceed by induction on the depth of the Böhm-tree. If the depth is 1, $K' = (x_{\sigma_1} : K_{\sigma_1}) \dots (x_{\sigma_n} : K_{\sigma_n})K_0$ and the lemma is obvious (P_i are variables and P is just a permutation).

Let x occur in the *kind* $K' \equiv (x'_{\sigma_1} : K'_{\sigma_1}) \dots (x'_{\sigma_n} : K'_{\sigma_n}) K'_0$. Does it imply that it occurs in the *kind* of z? As above,

- $P_1 : K_1$ (in appropriate context),
- $P_2 : [P_1/x_1]K_2$,
- $\dots$,
- $P_n : [P_{n-1}/x_{n-1}](\dots [P_1/x_1]K_n)$,
- $zP_1 \dots P_n : [P_n/x_n](\dots [P_1/x_1]K_0) = K'_0$.

There are three possibilities: (i) x occurs in the *kind* of z (and we are done); (ii) x occurs in one of K'_{σ_i}; (iii) x occurs in one of the P_i and into K'_0 (via substitution).

In case (ii) x occurs in *kind* of the head variable of P_{σ_i}. We may apply I.H. (for implication in opposite direction) and deduce that x occurs also in the *kind* of P_{σ_i}. We always may choose the leftmost of such P_{σ_j}, and conclude that a matching occurrence of x must exist in the *kind* of z.

In case (iii) we use Lemma 18 and arrive to the previous case.

Now, let us consider the opposite implication for P. Let x occur in the *kind* of z. Either it lies in K_0 (and then will occur in the *kind* of P as well) or it must be matched by the *kind* of one of P_i. Then by I.H. it occurs also in the *kind* of its head variable and into the prefix of P, and thus into K'. ◀

▶ **Corollary 21.** *If $\Gamma \vdash P : (K)K'$ is an isomorphism in LF (all is in normal form) the same free variables occur into K and K'.*

Proof. The term $e(P)$ has to be a c.f.h.p., so

$$P \equiv [z : (x_1 : K_1) \dots (x_n : K_n) K_0][x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n).$$

We apply previous lemma to $[x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n)$ in context $\Gamma, z : (x_1 : K_1) \dots (x_n : K_n) K_0$. ◀

Below we consider some properties of dependency of variables (relations $\lhd$ and $\lhd*$) that we shall use in our study of isomorphism.

▶ **Lemma 22.** *Let us consider $\Gamma \vdash P : (K)K'$, $\Gamma \vdash P' : (K')K$ such that*

- *$\Gamma \vdash P : (K)K'$, $\Gamma \vdash P' : (K')K$ are derivable in LF,*
- *$P \equiv [z : (x_1 : K_1) \dots (x_n : K_n) K_0][x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n)$,*
- *$P' \equiv [z' : (x_{\sigma_1} : K_{\sigma_1}) \dots (x_{\sigma_n} : K_{\sigma_n}) K'_0][x_1 : K_1] \dots [x_n : K_n](z' P'_{\sigma_1} \dots P'_{\sigma_n})$,*
- *and $e(P)$ and $e(P')$ are mutually inverse c.f.h.p.*

Then $x_i \lhd x_j$ iff $x'_i \lhd x'_j$.

Proof. Without loss of generality, we may consider also the terms $[x'_{\sigma_1} : K'_{\sigma_1}] \dots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \dots P_n)$ in the context $\Gamma, z : (x_{\sigma_1} : K_{\sigma_1}) \dots (x_{\sigma_n} : K_{\sigma_n}) K'_0$ and $x_1 : K_1] \dots [x_n : K_n](z' P'_{\sigma_1} \dots P'_{\sigma_n})$ in the context $\Gamma, z' : (x'_{\sigma_1} : K'_{\sigma_1}) \dots (x'_{\sigma_n} : K'_{\sigma_n}) K'_0$. Let us prove that $x_i \lhd x_j \Rightarrow x'_i \lhd x'_j$[4].

Because $e(P)$ is a c.f.h.p., the head variables of P_i are x'_i $(1 \leq i \leq n)$. Since $x_i \lhd x_j$, x_i occurs in K_j, the type of P_j (in appropriate context) is

$$[P_{j-1}/x_{j-1}](\dots [P_1/x_1]K_j),$$

[4] Let us emphasize that here the dependency between x_i and x_j, respectively x'_i and x'_j should be considered, not between x'_{σ_i} and x'_{σ_j}.

thus x'_i does occur in the *kind* of P_j. By Lemma 20 it occurs also into the *kind* of x'_j, and $x'_i \lhd x'_j$.

For opposite implication, we consider P'. ◀

▶ **Corollary 23.** *The partial order $\lhd*$ generated by $\lhd$ on $x_1, \ldots, x_n$ coincides with $\lhd*$ generated by $\lhd$ on $x'_1, \ldots, x'_n$.*

One more lemma:

▶ **Lemma 24.** *Let $P \equiv [x'_{\sigma_1} : K'_{\sigma_1}] \ldots [x'_{\sigma_n} : K'_{\sigma_n}](zP_1 \ldots P_n)$ as above. Then x'_i, the head variable of P_i, does not occur in $P_j, j < i$. Similar result holds for P'.*

Proof. The proof is based on the same idea as in Lemma 19. We consider the Böhm-tree of $e(P)$ and the ordering of the paths as above. We assume that x'_j does occur in (the *kind* of) some variable in P_j Then there is a smallest path leading to corresponding occurrence of an abstracted variable in the tree.

If it belongs to the prefix of P_j (the path has the length 1) then x'_j must belong to K_j in the type of z (because of minimality of the path it cannot come from substitution of P_l with $l < j$ into K_j), and we obtain a contradiction, because there is no occurrences of x'_j into *kind* of z (z lies more to the left in the context).

If the smallest path is longer, similar contradiction appears because we can show that an occurrence of x'_j must appear in the *kind* in the node that immediately precedes the end of this smallest path. ◀

The following lemma prepares the inductive step of our main theorem.

▶ **Lemma 25** (Decomposition). *Let $\Gamma \vdash P : (K)K'$, $\Gamma \vdash P' : (K')K$ be as in previous lemma. Let us consider the term*

$$R \equiv [z'' : (x''_1 : K''_1) \ldots (x''_n : K''_n)K''_0][x''_{\sigma_1} : K''_{\sigma_1}] \ldots [x''_{\sigma_n} : K''_{\sigma_n}](z''x''_1 \ldots x''_n).$$

Here R represents permutation. In particular, $K''_1 \equiv K'_1, K'' \equiv [x''_1/x'_1]K'_1, \ldots, K''_n \equiv [x''_{n-1}/x'_{n-1}] \ldots [x''_1/x'_1]K'_n$, $K''_0 \equiv [x''_{n-1}/x'_{n-1}] \ldots [x''_1/x'_1]K'_0$

Consider also

$$P_0 \equiv [z : (x_1 : K_1) \ldots (x_n : K_n)K_0][x'_1 : K'_1] \ldots [x'_n : K'_n](zP_1 \ldots P_n)$$

and

$$P'_0 \equiv [z' : (x'_1 : K'_1) \ldots (x'_n : K'_n)K'_0][x_1 : K_1] \ldots [x_n : K_n](z'P'_1 \ldots P'_n).$$

Then $\Gamma \vdash R : (K'')K'$, $\Gamma \vdash P_0 : (K)K''$, $\Gamma \vdash P'_0 : (K'')K$ are derivable in LF and the following decompositions hold:

- $\Gamma \vdash P = R \circ P_0 : (K)K'$,
- $\Gamma \vdash P' \circ R = P'_0 : (K'')K$.

Proof. The derivability of all these terms relies on Lemma 22 and its Corollary. Verification of equalities uses standard reductions. For example, let us consider $\Gamma \vdash P = R \circ P_0 : (K)K'$.

Composition of two terms is defined as usual: $R \circ P_0 \equiv [z : K](R(P_0 z))$. By two β-reductions we obtain

$$[z : K][x''_{\sigma_1} : K''_{\sigma_1}] \ldots [x''_{\sigma_n} : K''_{\sigma_n}]([x_1 : K_1] \ldots [x_n : K_n](zP_1 \ldots P_n)x''_1 \ldots x''_n).$$

After that follows the series of β-reductions and renaming of bound variables ($x'' \to x'$) that gives P.

Verification for another equality is similar. ◀

Proof of the Theorem 10 (continuation).

Let $\Gamma \vdash P : (K)K'$ be an isomorphism. Then there exists its inverse $\Gamma \vdash P'(K')K$. In particular, $\Gamma \vdash P' \circ P = [z : K]z : (K)K$.

Using the Decomposition Lemma, we may write

$$\Gamma \vdash P' \circ (R \circ P_0) = (P' \circ R) \circ P_0 = P'_0 \circ P_0 = [z : K]z : (K)K.$$

Similar fact holds for $P \circ P'$.

Via two standard β-reductions we obtain

$$P'_0 \circ P_0 = [z : K][x_1 : K_1] \dots [x_n : K_n]([x'_1 : K'_1] \dots [x'_n : K'_n](zP_1 \dots P_n)P'_1 \dots P'_n).$$

Before we continue with reductions, let us see what can be established about contexts and *kinding* of $P_1, \dots, P_n$ and $P'_1, \dots, P'_n$[5].

Consider now the typing of $P_1, \dots, P_n$ and $P'_1, \dots, P'_n$. A straightforward use of properties of LF-derivations gives us:

$$\Gamma, z : K, x'_1 : K'_1, \dots, x'_n : K'_n \vdash P_1 : K_1,$$

$$\Gamma, z : K, x'_1 : K'_1, \dots, x'_n : K'_n \vdash P_2 : [P_1/x_1]K_2,$$

$$\dots$$

$$\Gamma, z : K, x'_1 : K'_1, \dots, x'_n : K'_n \vdash P_n : [P_{n-1}/x_{n-1}](\dots [P_1/x_1]K_n),$$

respectively,

$$\Gamma, z' : K', x_1 : K_1, \dots, x_n : K_n \vdash P'_1 : K'_1,$$

$$\Gamma, z' : K', x_1 : K_1, \dots, x_n : K_n \vdash P'_2 : [P'_1/x_1]K'_2,$$

$$\dots$$

$$\Gamma, z' : K', x_1 : K_1, \dots, x_n : K_n \vdash P'_n : [P'_{n-1}/x_{n-1}](\dots [P'_1/x_1]K'_n).$$

The derivability of these judgements is obtained using the known properties of LF-derivations (see [12, 14]).

Using Corollary 19, Lemma 24, and then applying Lemma 1 (strengthening), we can make the contexts considerably smaller:

$$\Gamma, x'_1 : K'_1 \vdash P_1 : K_1,$$

$$\Gamma, x'_1 : K'_1, x'_2 : K'_2 \vdash P_2 : [P_1/x_1]K_2,$$

$$\dots$$

$$\Gamma, x'_1 : K'_1, \dots, x'_n : K'_n \vdash P_n : [P_{n-1}/x_{n-1}](\dots [P_1/x_1]K_n),$$

respectively,

$$\Gamma, x_1 : K_1 \vdash P'_1 : K'_1,$$

$$\Gamma, x_1 : K_1, x_2 : K_2 \vdash P'_2 : [P'_1/x_1]K'_2,$$

$$\dots$$

$$\Gamma, x_1 : K_1, \dots, x_n : K_n \vdash P'_n : [P'_{n-1}/x_{n-1}](\dots [P'_1/x_1]K'_n).$$

[5] This is important, because as the Example 8 shows x'_1 may very well occur into $P_2, \dots, P_n$, x'_2 occur into $P_3, \dots, P_n$, *etc.*

It is easily verified that the assumption that P_0 and P'_0 are mutually inverse implies that P_j and P'_j are mutually inverse (in the above contexts). Notice that the rank is not changed by substitution (Remark 12), so the ranks of *kinds* of $[x'_1 : K'_1]P_1, \ldots, [x'_n : K'_n]P_n$, $[x_1 : K_1]P'_1, \ldots, [x_n : K_n]P'_n$ are strictly smaller than the ranks of $(K)K'$ and $(K')K$. We may apply inductive hypothesis to the pairs $P_1, P'_1, \ldots, P_n, P'_n$[6]. So, the number of the isomorphisms of the form P_0, P'_0 is finite and we may list them using isomorphisms corresponding to kinds of smaller rank, obtained by inductive hypothesis.

To pass to the general case, we have to include permutations represented by R. Their number is also finite. The upper bound is given by the number of permutations on $\{1, \ldots, n\}$ and actual number may be less due to the constraints imposed by the relation $\lhd*$ of variable dependency. Of course they all can be listed constructively, and so all the isomorphisms for a given K may be listed. ◀

▶ **Corollary 26.** *The relation of isomorphism of kinds in LF is decidable.*

Proof. An algorithm (not very efficient) works as follows. Let $\Gamma \vdash K\mathbf{kind}, \Gamma \vdash K'\mathbf{kind}$. Using main theorem, we create the list of all *kinds* that are isomorphic to K and verify whether any of them is equal to K' (*e.g.*, reducing to normal form). ◀

▶ Remark 27. When $P : (K)K'$ is an isomorphism, the $rank(K)$ permits to obtain an upper bound to the depth of the Böhm's tree of the f.h.p. $e(P)$ and this in its turn may be used to obtain an upper bound on the number of isomorphisms in the theorem.

4 Conclusion

The "core system" that we studied, Z. Luo's LF with variables, *kind* **Type** and dependent product (and definitional equality) is relatively limited, but the limited character of this system permits to obtain a complete deciding algorithm for isomorphism relation between *kinds* in spite of the fact that local (or non-contextual) rewriting does not work. The limited character of this system permits also to describe completely the structure of isomorphisms. Whether the term P is an isomorphism turns out to be decidable as well.

The restrictions on isomorphisms imposed by type-dependencies allow more (not less) "fine-tuning" than in the case of simply-typed λ-calculus. Isomorphisms of kinds to themselves are called *automorphisms*. We have a sketch of a proof (work in progress) that every finite group is isomorphic to the group of automorphisms of some kind in LF. The groups of automorphisms of simple types correspond to automorphisms of finite trees. An arbitrary finite group can not be represented in this way.

The main use of LF is to specify other type theories. To do this, LF is extended by new constants, rules for these constants, *etc.* For example the Second Order Logic SOL and Universal Type Theory UTT are defined in [11]. There are other possibilities to build type theories using LF, *e.g.*, on may add new equality rules, like the analog of the axiom C from [10]. Another modification of equality (for the whole UTT) was studied in [13].

The isomorphisms in LF described above will remain isomorphisms in these type theories but, of course, other isomorphisms may appear. The study of these isomorphisms remains an open problem.

[6] In this order, because the isomorphisms obtained by inductive hypothesis are substituted into *kinds* more to the right.

We did not yet study (and try to improve) the complexity of decision procedures for isomorphism relation between *kinds* and for the property of a term P to be an isomorphism.

All this is left for study in the near future.

Acknowledgements. The author would like to express his thanks to the organizers of the Univalent Foundations Year at IAS, Princeton (September 2012 – August 2013), the organizers of the Trimester on Semantics of Proofs and Certified Mathematics at the Institute Henri Poincaré (Paris, April–July 2014) who partly supported his participation in these research programs and created excellent work environment, and to anonymous referees for their valuable remarks.

References

1 K. Bruce, R. Di Cosmo, and G. Longo. Provable isomorphisms of types. Math. Str. in Comp. Science, 2(2):231–247, 1992.
2 D. Chemouil.Isomorphisms of simple inductive types through extensional rewriting. Math. Str. in Comp. Science, 15(5):875–915, 2005.
3 R. Di Cosmo. Isomorphisms of types: from lambda-calculus to information retrieval and language design. Birkhäuser, 1995.
4 The Coq Reference Manual, `http://coq.inria.fr/coq/refman/`.
5 D. Delahaye. Information Retrieval in a Coq Proof Library Using Type Isomorphisms. TYPES 1999: 131-147
6 M. Fiore. Isomorphisms of generic recursive polynomial types. In: POPL'04, pp. 77–88, New York, NY, USA, 2004. ACM
7 M. Fiore, R. Di Cosmo and V. Balat. Remarks on isomorphisms in typed lambda calculi with empty and sum types. Annals of Pure and Applied Logic, 141(1):35–50, 2006.
8 H. Goguen. A Typed Operational Semantics for Type Theory. PhD thesis, University of Edinburgh, 1994.
9 The HoTT Book. Homotopy Type Theory: Univalent Foundations of Mathematics. IAS Princeton, 2013.
10 G. Longo, K. Milsted and S. Soloviev. The Genericity Theorem and effective Parametricity in Polymorphic lambda-calculus. Theor. Comp. Science, 121(1993), pp. 323–349.
11 Z. Luo. Computation and Reasoning. A Type Theory for Computer Science. Oxford, 1994.
12 Z. Luo, S. Soloviev, T. Xue. Coercive subtyping: Theory and implementation. Informaion and Computation, 223 (2013), pp. 18–42.
13 L. Marie-Magdeleine. Sous-typage coercitif en présence de réductions non-standards dans un système aux types dépendants. Thèse de doctorat. Université Toulouse 3 Paul Sabatier, 2009.
14 S. Soloviev, Z. Luo. Coercion completion and conservativity in coercive subtyping. Annals of Pure and Applied Logic, 113(1–3):297–322, 2002.
15 S. V. Solov'ev. The category of finite sets and cartesian closed categories. J. of Soviet Mathematics, 22(3):1387–1400, 1983.
16 C. Stirling. Proof Systems for Retracts in Simply Typed Lambda Calculus. ICALP (2) 2013, pp. 398–409.

An Intuitionistic Analysis of Size-change Termination

Silvia Steila

Dipartimento di Informatica, Università degli studi di Torino
Corso Svizzera 185 Torino, Italy
steila@di.unito.it

Abstract

In 2001 Lee, Jones and Ben-Amram introduced the notion of size-change termination (SCT) for first order functional programs, a sufficient condition for termination. They proved that a program is size-change terminating if and only if it has a certain property which can be statically verified from the recursive definition of the program. Their proof of the size-change termination theorem used Ramsey's Theorem for pairs, which is a purely classical result. In 2012 Vytiniotis, Coquand and Wahlsteldt intuitionistically proved a classical variant of the size-change termination theorem by using the Almost-Full Theorem instead of Ramsey's Theorem for pairs. In this paper we provide an intuitionistic proof of another classical variant of the SCT theorem: our goal is to provide a statement and a proof very similar to the original ones. This can be done by using the H-closure Theorem, which differs from Ramsey's Theorem for pairs only by a contrapositive step. As a side result we obtain another proof of the characterization of the functions computed by a tail-recursive SCT program, by relating the SCT Theorem with the Termination Theorem by Podelski and Rybalchenko. Finally, by investigating the relationship between them, we provide a property in the "language" of size-change termination which is equivalent to Podelski and Rybalchenko's termination.

1998 ACM Subject Classification F.4.1 Mathematical Logic, F.3.1 Specifying and Verifying and Reasoning about Programs

Keywords and phrases Intuitionism, Ramsey's Theorem, Termination

Digital Object Identifier 10.4230/LIPIcs.TYPES.2014.288

1 Introduction

An important topic in theoretical computer science is determining whether a program is terminating on a given input by studying its source code. Even if in general this problem, known as Halting problem, is undecidable, in some special cases this can be done. In 2001 Lee, Jones and Ben-Amram introduced the notion of size-change termination. A first order functional program $\mathcal{P}$ is SCT if for any infinite sequence of calls which follows the control of $\mathcal{P}$ there exists a variable whose value has to decrease infinitely many times. If the domain of the values of $\mathcal{P}$ is well-founded this condition guarantees the termination. In [14] the authors prove that any first order functional program is SCT if and only if it satisfies some combinatorial property which can be statically verified from the recursive definition of the program. We will call this result the SCT Theorem. In order to prove this theorem the authors use Ramsey's Theorem for pairs [19]. This result states that given any coloring over the edges of the complete graph with infinitely many nodes in finitely many colors, there exists an infinite "homogeneous" set. A set of nodes of a colored graph is said "homogeneous" when any two elements of the set are connected with the same color. It is well-known that Ramsey's Theorem for pairs is not an intuitionistically valid result. To be precise, we need

20th International Conference on Types for Proofs and Programs (TYPES 2014).
Editors: Hugo Herbelin, Pierre Letouzey, and Matthieu Sozeau; pp. 288–307

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

the law of excluded middle for Σ^0_3 formulas in order to prove the fragment of this theorem which can be expressed as a schema of first order statements [5].

The SCT Theorem is not the only result which characterizes the termination of some class of programs by using Ramsey's Theorem for pairs. In 2004 Podelski and Rybalchenko introduced the notion of transition invariant. T is a transition invariant for a transition-based program $\mathcal{R}$ if T contains the transitive closure of the transition relation of $\mathcal{R}$. In [18] the authors proved their Termination Theorem: a while-if program is terminating if and only if there exists a transition invariant which satisfies certain properties. We refer to Section 2 for details. The proof of the Termination Theorem uses Ramsey's Theorem for pairs, as it is the case for the SCT Theorem.

The Termination Theorem is intuitionistically provable if we consider inductive well-foundedness instead of well-foundedness (see Section 2). Intuitionistic proofs of this theorem are given in [20] and [6]. In both the proofs Ramsey's Theorem for pairs is replaced by some intuitionistic result. In the first one it is replaced by the Almost-full Theorem [9] by Coquand, while in the second one it is replaced by the H-closure Theorem [6]. Both of them are classically (but not intuitionistically) equivalent to Ramsey's Theorem for pairs. They keep the combinatorial strength of Ramsey's Theorem for pairs required in the proof of the Termination Theorem but they drop the classical part.

In this paper we prove some intuitionistic version of the SCT Theorem. From the intuitionistic proof we extract an upper bound for the number of steps needed by an SCT program to terminate. In Section 3 we define a variant of SCT, which we call SCT* and which is classically equivalent but intuitionistically more informative. SCT* is defined by taking a contrapositive and, as in the case of the Termination Theorem, by using the inductive version of well-foundedness. Thanks to it, in Section 4, we may prove the SCT Theorem without using classical principles. We will use the H-closure Theorem instead of Ramsey's Theorem for pairs. This is not the first intuitionistic proof of the SCT Theorem since Vytiniotis, Coquand and Wahlsteldt in [20] intuitionistically proved this result by using the Almost-Full Theorem. However since we find no way to intuitionistically deduce the H-closure Theorem from the Almost-Full Theorem, there are no apparent relationships between the proofs.

In [3] Ben-Amram proved that any tail-recursive SCT functional program is primitive recursive. In Section 5 we give a completely different proof of this result based on the bounds found for the Termination Theorem in [4] and [11]. We can use these bounds since the SCT Theorem and the Termination Theorem are strictly related. Heizmann, Jones and Podelski proved that size-change termination is a property strictly stronger than termination [13]. By applying an argument similar to the one used in [13] we will get the bound for a tail-recursive SCT program from the one for the Termination Theorem provided in [11]. As a corollary of the results presented in [13], in Section 6, we find a property in the "language" of size-change termination which is equivalent to the termination with transition invariants.

In this paper we will work in Heyting Arithmetic (HA); all the proofs are intuitionistic.

2 Transition Invariants and the H-closure Theorem

In this section we summarize the main definitions and properties of transition invariants. Transition invariants are used by Podelski and Rybalchenko in [18] in order to characterize terminating programs. The Termination Theorem by Podelski and Rybalchenko states that a transition-based program $\mathcal{R}$ is terminating if and only if there exists a disjunctively well-founded transition invariant for $\mathcal{R}$. The proof of this result was classical since the authors use Ramsey's Theorem for pairs in order to prove it. However we can modify the

definition of termination with a classically equivalent one and show that the theorem is intuitionistically true ([20] and [6]).

2.1 Transition Invariants

In this subsection we recall the definition of transition invariant and the Termination Theorem. For all details we refer to [18].

▶ **Definition 1** (Transition Invariants). As in [18]:

- A transition-based program $\mathcal{R} = (S, I, R)$ consists of:
  - S: a set of states,
  - I: a set of initial states, such that $I \subseteq S$,
  - R: a transition relation, such that $R \subseteq S \times S$.
- A computation is a maximal sequence of states $s_0, s_1, \ldots$ such that
  - $s_0 \in I$,
  - $(s_{i+1}, s_i) \in R$ for all $i \in \mathbb{N}$.
- The set Acc of accessible states consists of all states that appear in some computation.
- The transition-based program $\mathcal{R}$ is terminating if and only if $R \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded.
- A transition invariant T is a set which contains the transitive closure of the transition relation R restricted to the accessible states Acc. Formally,

  $$R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \subseteq T.$$

- A relation T is disjunctively well-founded if it is a finite union $T = T_0 \cup \cdots \cup T_{n-1}$ of well-founded relations.

Being well-founded is not preserved under binary unions, therefore a disjunctively well-founded relation can be ill-founded. We represent each state as a finite map s which provides the values of the variables and the location of s (for an introduction to these concepts see [13, pag 8]). Given a state s and a variable x we will write $s(x)$ to mean the value of x in the state s, while $s(\mathrm{pc})$ is the current location of s.

The main result by Podelski and Rybalchenko is the following.

▶ **Theorem 2** (Termination Theorem, Theorem 1 [18]). *The transition-based program $\mathcal{R}$ is terminating if and only if there exists a disjunctively well-founded transition invariant for $\mathcal{R}$.*

By unfolding definitions Theorem 2 states that a binary relation R is well-founded if and only if there exist a natural number n and n-many well-founded relations $R_0, \ldots, R_{n-1}$ whose union contains the transitive closure of R. This is non trivial since in general a disjunctively well-founded relation can be ill-founded. The fact that a transitive binary relation which is the union of two well-founded relations is well-founded has been remarked before Podelski and Rybalchenko (for instance see [12, pag 31]).

Let us see as example one simple application of the Termination Theorem.

▶ **Example 3.** Examine the following transition-based program which computes $\exp = x^y$ whenever $x, y > 0$ and $\exp = 1$ at the beginning.

$$
\begin{array}{l}
\text{while}(y > 0) \\
\quad \{ \\
\quad l : \text{temp} = 0; z = x; \\
\quad \text{while}(z > 0) \\
\qquad \{\text{temp} = \text{temp} + \exp; z = z - 1; \} \\
\quad l' : y = y - 1; \exp = \text{temp}; \\
\quad \}
\end{array}
$$

This program terminates since there exists a disjunctively well-founded transition invariant for it:

$$T := \{(s', s) \mid s'(z) < s(z)\} \cup \{(s', s) \mid s'(y) < s(y)\} \cup \{(s', s) \mid s'(\text{pc}) = l \ \wedge \ s(\text{pc}) = l'\}.$$

Observe that, in this case, it would not be difficult to directly prove termination by using the lexicographic ordering.

From the Termination Theorem Cook, Podelski and Rybalchenko extracted an algorithm which produces disjunctively well-founded transition invariants for some terminating programs [8], as in the example above.

2.2 Overview on inductive well-foundedness

In this subsection we recall the main results about well-foundedness which we require in this paper. All the results presented in this subsection are presented in previous work [6], we refer to it for details. From now on we consider the inductive definition of well-foundedness as in [1, 2], which is classically equivalent to the usual one if we assume the Axiom of Dependent Choice.

Classically a binary relation R over a set S is well-founded if there are no infinite decreasing R-chains. We say that $x \in S$ is classically R-well-founded if there are no infinite decreasing R-chains from x. Here we are interested in the inductive definition of well-foundedness, which is classically, but not intuitionistically, equivalent to the classical one. Hence we say that a binary relation R over S is inductively well-founded if and only if every element of S belongs to any R-inductive property X:

$$\forall x \forall X((\forall y((\forall z(zRy \implies z \in X)) \implies y \in X)) \implies x \in X).$$

We say that $x \in S$ is inductively R-well-founded if $x \in X$ holds for any R-inductive property X. For short, when clear from the context, we will say well-founded instead of inductive well-founded.

An important tool for proving that a relation is well-founded are simulations [16]. A simulation relation is a binary relation which connects two other binary relations. Intuitively a simulation of a binary relation $R \subseteq S^2$ into a binary relation $R' \subseteq S'^2$ is a way of associating, step by step, any R-decreasing sequence to some R'-decreasing sequence.

▶ **Definition 4.** Let R be a binary relation on S and R' be a binary relation on S'. Let U be a binary relation on $S \times S'$, and let $\circ$ denote the composition between two relations.

- U is a simulation of R in R' if and only if $R \circ U \subseteq U \circ R'$; i.e.

$$\forall x, z \in S\ \forall y \in S'\ ((xUy \wedge zRx) \implies \exists w \in S'\ (wR'y \wedge zUw))$$

- A simulation relation U of R in R' is total if $\mathrm{dom}(U) = \{x \in S \mid \exists y \in S'(xUy)\} = S$.
- R is simulable in R' if there exists a total simulation relation U of R in R'.

The next proposition shows that the simulation relations "preserve" inductively well-foundedness: it is a generalization of the preservation of well-foundedness by inverse image [17].

▶ **Proposition 5.** *Let R be any binary relation on S, and let R' be a binary relation on S'.*

1. *For any $x \in S$:*

$$x \text{ is } R\text{-well-founded} \iff \forall y(yRx \implies y \text{ is } R\text{-well-founded}).$$

2. *If U is a simulation of R in R' and if xUy and y is R'-well-founded, then x is R-well-founded.*
3. *If U is a simulation of R in R' and R' is well-founded, then $\mathrm{dom}(U)$ is R-well-founded.*
4. *If R is simulable in R' and R' is well-founded, then R is well-founded.*

Let R be a binary relation over a set S. We say that a function $f : S \to \mathbb{N}$ is a weight function if for any $x,\ y \in S$

$$xRy \implies f(x) < f(y).$$

R has height ω if and only if R has a weight function.

By using the total simulation $U = \{(x, f(x)) \mid x \in \mathrm{dom}(R)\}$ we can easily prove that if R has height ω then it is inductively well-founded. Moreover if R is inductively well-founded and finitely branching, then $f(x) = \sup\{f(y) + 1 \mid yRx\}$ is a weight function for R.

We also need to recall a well-known result about finite relations. Let R be a binary relation on any finite set $\{x_i \mid i \leq k\}$. A R-cycle is a finite sequence $\langle x_{i_0}, \ldots, x_{i_n}\rangle$ for some $n \in \mathbb{N}$, such that and $i_j \leq k$ for any $j \leq n$, and

$$x_{i_0} = x_{i_n} R x_{i_{n-1}} R \ldots R x_{i_1} R x_{i_0}.$$

If $n = 0$ we ask that $x_{i_0} R x_{i_0}$ and we call x_{i_0} a loop of R.

▶ **Proposition 6.** *Let R be any binary relation on a finite set S.*

- *R is well-founded if and only if there are no R-cycles.*
- *If R is a strict order then R is well-founded.*

2.3 H-closure Theorem

The H-closure Theorem, where H stands for "homogeneous", is the result used in [6] to give an intuitionistic proof of the Termination Theorem. The statement of the H-closure Theorem was obtained from Ramsey's Theorem for pairs by taking a contrapositive and it is intuitionistically provable. The two theorems are equivalent in RCA_0 [7], the base system of Reverse Mathematics which consists of recursive comprehension and Σ^0_1-induction. We may define H-closure as follows.

Let $\succ$ denote the one-step expansion between finite sequences; i.e.

$$\langle y_0, \ldots, y_{m-1}\rangle \succ \langle x_0, \ldots, x_{n-1}\rangle \iff m = n + 1 \wedge \forall i < n(x_i = y_i).$$

We call an R-homogeneous sequence any finite transitive decreasing R-chain. We say that R is homogeneous-well-founded, just H-well-founded for short, if the relation $\succ$ on the set of R-homogeneous sequences is well-founded. Being H-well-founded is weaker than being well-founded, as we will see in a moment. Formally, the definition runs as follows.

▶ **Definition 7** (H-well-foundedness)**.** Let R be a binary relation on S.

- $H(R)$, the set of R-homogeneous sequences, is the set of the R-decreasing transitive finite sequences on S:

$$\langle x_0, \dots, x_{n-1} \rangle \in H(R) \iff \forall i, j < n \ (i < j \implies x_j R x_i).$$

- R is H-well-founded if $H(R)$ is $\succ$-well-founded.

From the previous definition follows that R is classically H-well-founded if and only if there are no infinite decreasing transitive R-chains. It also follows that if R is decidable then also $H(R)$ is. If S is finite, then we may describe the difference between well-foundedness and H-well-foundedness as follows: R is well-founded if and only if R has no cycles, while R is H-well-founded if and only if R has no loops (there is no $x \in S$ such that xRx). There is a strong connection between well-foundedness and H-well-foundedness, described in the following proposition.

▶ **Proposition 8** (Proposition 1 [6])**.** *Let R be a binary relation.*

1. *If R is well-founded then R is H-well-founded.*
2. *If R is H-well-founded and R is transitive then R is well-founded.*

A last example. Consider $R = \{(n+1, n) \mid n \in \mathbb{N}\}$. It is straightforward to check that it is not well-founded and it is H-well-founded since

$$H(R) = \{\langle\rangle\} \cup \{\langle n \rangle \mid n \in \mathbb{N}\} \cup \{\langle n, n+1 \rangle \mid n \in \mathbb{N}\}.$$

The H-closure theorem states that H-well-foundedness is closed under finite unions. Formally

▶ **Theorem 9** (H-closure Theorem, Theorem 2 [6])**.** *Let $R_0, \dots, R_{n-1}$ be binary relations. If $R_0, \dots, R_{n-1}$ are H-well-founded then $R_0 \cup \dots \cup R_{n-1}$ is H-well-founded.*

Classically, and if we take a contrapositive, the H-closure Theorem states: if there is some infinite $R_0 \cup \dots \cup R_{n-1}$-homogeneous sequence then for some $i < n$ there is some infinite R_i-homogeneous sequence. From this remark we may classically check that the H-closure Theorem is but a variant of Ramsey's Theorem for pairs [19]. However, the H-closure Theorem is intuitionistically provable, while Ramsey's Theorem for pairs is not [6].

In [6] from the H-closure Theorem we obtained an intuitionistic proof of the Termination Theorem. Furthermore, by analysing this intuitionistic proof, in [4] the authors got a characterization of the Termination Theorem via the primitive recursive functions.

▶ **Theorem 10.** *Assume that the transition relation of the program $\mathcal{R}$ is the graph of a primitive recursive function restricted to a primitive recursive subset. If $\mathcal{R}$ has a disjunctively well-founded transition invariant whose relations are primitive recursive and have height ω then it computes a primitive recursive function.*

For short we say that a transition invariant has height ω if the relations which compose it have height ω. In [11] there is another proof of this result based on the Dickson Lemma. With both approaches we can characterize the level of the primitive recursive hierarchy

reached by the transition-based program $\mathcal{R}$. We denote with $\mathcal{F}_k$ the usual k-class of the Fast Growing Hierarchy [15]. Define

$$F_0(x) = x + 1$$
$$F_{k+1}(x) = F_k^{(x+1)}(x).$$

Then $\mathcal{F}_k$ is the closure under limited recursion and substitution of the set of functions composed of constant, projections, sum and F_h for any $h \leq k$.

As shown in [11] if $\mathcal{R}$ is a program such that

- its transition relation R is the graph of a function in $\mathcal{F}_2$;
- there exists a transition invariant for $\mathcal{R}$ which is the union of k relations, all having weight functions in $\mathcal{F}_1$.

Then the function computed by $\mathcal{R}$ is in $\mathcal{F}_{k+1}$.

3 From SCT to SCT*

In this section after a brief summary on the original definition of SCT presented in [13], we introduce a variant of SCT, which we call SCT*, and which is classically equivalent to SCT. Thanks to this definition we can intuitionistically prove the SCT Theorem. This is similar to what we did in the previous section: in order to intuitionistically prove the Termination Theorem we had to consider a classical equivalent of the termination property which is intuitionistically easier to prove. We obtain SCT* from SCT by taking the contrapositive and by considering the inductive well-foundedness instead of the classical one.

From now on we will deal with a language for functions on $\mathbb{N}$ with call-by-value semantics considered in [14]. We use the recursive definitions and notations for maps $f : \mathbb{N}^n \to \mathbb{N}$ which Heizmann, Jones and Podelski present in their paper, for details see [13, pages 2-4]. Another useful reference is [3].

3.1 Size-change Termination

Here we recall the definition of SCT. If the reader is familiar with this definition, he may skip this subsection.

Informally, a recursive definition of a function has the SCT property if in every infinite sequence of function calls there is some infinite sequence of values of arguments which is weakly decreasing, and strictly decreasing infinitely many times. In the case the domain of the function is $\mathbb{N}$, there is no such sequence of values, and SCT is a sufficient condition for termination. In order to formally express SCT, first of all we need the definition of size-change graph. From now on we fix a recursive definition for a program $\mathcal{P}$ characterized as above. Let f be defined in $\mathcal{P}$ as follows:

$$f(x_0, \ldots, x_{n-1}) := \text{if}(\ldots)\ \text{then}\ f_1(\ldots)\ \text{else if}(\ldots)\ \text{then}\ f_2(\ldots)\ \text{else}\ f_3(\ldots).$$

Then we will denote the set $\{x_0, \ldots, x_{n-1}\}$ by $\text{Var}(f)$. Given such f, a state is a pair $(f, \mathbf{v})$ where $\mathbf{v}$ is a finite sequence of natural numbers whose length is n. If in the definition of f there is a call

$$\ldots \tau : g(e_0, \ldots, e_{m-1})$$

we define a state transition $(f, \mathbf{v}) \xrightarrow{\tau} (g, \mathbf{u})$ to be a pair of states such that $\mathbf{u}$ is the sequence of values obtained by the expressions $(e_0, \ldots, e_{m-1})$ when f is evaluated with $\mathbf{v}$.

▶ **Definition 11** (Size-change graph)**.** Let f, g be defined in $\mathcal{P}$. A size-change graph $G : f \to g$ for $\mathcal{P}$ is a bipartite directed graph on $(\mathrm{Var}(f), \mathrm{Var}(g))$. The set of edges is a subset of $\mathrm{Var}(f) \times \{\downarrow, \Downarrow\} \times \mathrm{Var}(g)$ such that there is at most one edge for any $x \in \mathrm{Var}(f)$, $y \in \mathrm{Var}(g)$. We say that f is the source function of G and g is the target function of G.

We call $(x, \downarrow, y)$ the decreasing edge, and we denote it with $x \xrightarrow{\downarrow} y$. We call $(x, \Downarrow, y)$ the weakly-decreasing edge, and we denote it with $x \xrightarrow{\Downarrow} y$.

▶ **Definition 12.** Let $f(x_0, \ldots, x_{n-1})$ be defined with a call $\tau : g(e_0, \ldots, e_{m-1})$ (where $\mathrm{Var}(g) = \{y_0, \ldots, y_{m-1}\}$).

- The edge $x_i \xrightarrow{r} y_j$ safely describes the $x_i - y_j$ relation in the call τ, if for any $\mathbf{v} \in \mathbb{N}^n$ and $\mathbf{u} \in \mathbb{N}^m$ such that $(f, \mathbf{v}) \xrightarrow{\tau} (g, \mathbf{u})$, then $r = \downarrow$ implies that $u_j < v_i$ and $r = \Downarrow$ implies that $u_j \leq v_i$.
- The size-change graph G_τ is safe for the call τ if every edge in G_τ is a safe description.
- Set $\mathcal{G}_\mathcal{P} = \{G_\tau \mid \tau \text{ is a call in } P\}$. We say that $\mathcal{G}_\mathcal{P}$ is a safe description of $\mathcal{P}$ if for any call τ, G_τ is safe.

Note that the absence of edges between two variables x and y in the size-change graph G_τ which is safe for τ indicates either an unknown or an increasing relation in the call τ.

▶ **Definition 13.** A multipath $\mathcal{M}$ is a graph sequence $G_0, \ldots, G_n, \ldots$ such that the target function of G_i is the source function of G_{i+1} for any i. A thread is a connected path of edges in $\mathcal{M}$ that starts at some G_t, where $t \in \mathbb{N}$. A multipath $\mathcal{M}$ has infinite descent if some thread in $\mathcal{M}$ contains infinitely many decreasing edges.

▶ **Definition 14** (SCT program)**.** Let $\mathcal{T}$ be the set of calls in program $\mathcal{P}$. Suppose that each size-change graph $G_\tau : f \to g$ is safe for every call τ in

$$\mathcal{G}_\mathcal{P} = \{G_\tau \mid \tau \in \mathcal{T}\}.$$

$\mathcal{P}$ is size-change terminating (SCT) if, for any infinite call sequence $\pi = \tau_1, \ldots, \tau_n, \ldots$ that follows $\mathcal{P}$'s control flow, the multipath $M_\pi = G_{\tau_1}, \ldots, G_{\tau_n}, \ldots$ has an infinite descent.

3.2 Composing size-change graphs

As in [13], given two size-change graphs $G_0 : f \to g$ and $G_1 : g \to h$ we define their composition $G_0; G_1 : f \to h$. The composition of two edges $x \xrightarrow{\Downarrow} y$ and $y \xrightarrow{\Downarrow} z$ is one edge $x \xrightarrow{\Downarrow} z$. In all other cases the composition of two edges from x to y and from y to z is the edge $x \xrightarrow{\downarrow} z$. Formally, $G_0; G_1$ is the size-change graph with the following set of edges:

$$\begin{aligned} E = \{x \xrightarrow{\downarrow} z \mid \exists y \in \mathrm{Var}(g)\ \exists r \in \{\downarrow, \Downarrow\}\ ((x \xrightarrow{\downarrow} y \in G_0 \wedge y \xrightarrow{r} z \in G_1) \\ \vee (x \xrightarrow{r} y \in G_0 \wedge y \xrightarrow{\downarrow} z \in G_1))\} \\ \cup \{x \xrightarrow{\Downarrow} z \mid \exists y \in \mathrm{Var}(g)(x \xrightarrow{\Downarrow} y \in G_0 \wedge y \xrightarrow{\Downarrow} z \in G_1) \wedge \forall y \in \mathrm{Var}(g) \\ \forall r, r' \in \{\downarrow, \Downarrow\}\ ((x \xrightarrow{r} y \in G_0 \wedge y \xrightarrow{r'} z \in G_1) \implies r = r' = \Downarrow)\}. \end{aligned}$$

Observe that the composition operator ";" is associative. Given a finite call sequence $\pi = \tau_0, \ldots, \tau_{n-1}$ we define $G_\pi = G_{\tau_0}, \ldots, G_{\tau_{n-1}}$. Moreover we say that the size-change graph G is idempotent if $G; G = G$.

Given a finite set of size-change graphs $\mathcal{G}$, $\mathsf{cl}(\mathcal{G})$ is the smallest set which contains $\mathcal{G}$ and is closed by composition. Formally $\mathsf{cl}(\mathcal{G})$ is the smallest set such that

- $\mathcal{G} \subseteq \mathsf{cl}(\mathcal{G})$;
- If $G_0 : f \to g$ and $G_1 : g \to h$ are in $\mathsf{cl}(\mathcal{G})$, then $G_0; G_1 \in \mathsf{cl}(\mathcal{G})$.

Once we fixed the number of variables, there are only finitely many bipartite graphs with two labels for the edges, therefore classically $\mathsf{cl}(\mathcal{G})$ is finite. Moreover we can intuitionistically prove that it is finite thanks to the following proposition.

▶ **Proposition 15.** *Assume that S is a finite set where the equality is decidable and that op: $S \times S \to S$ is a computable map. Then the closure of any finite subset of S is intuitionistically finite.*

In fact if $I \subseteq S$, we can define $I_0 = I$, $I_{k+1} = \{\mathrm{op}(a,b) \mid a, b \in \bigcup\{I_h \mid h \leq k\}\} \setminus \bigcup\{I_h \mid h \leq k\}$. By decidability of equality, we may effectively compute $A \setminus B$ for any A, B finite subsets of S. Therefore we may intuitionistically prove by induction over $S \setminus \bigcup\{I_h \mid h \leq k\}$ that there is a $k \leq |S|$ such that $I_{k+1} = \emptyset$. Thus k defines the closure of I.

3.3 Definition of SCT*

As seen above, $\mathcal{P}$ is SCT if and only if

for any infinite call sequence π that follows $\mathcal{P}$, M_π has an infinite descent.

Now we want to apply some classical step in order to obtain a statement SCT* classically equivalent to SCT but intuitionistically easier to prove. From the definition of SCT, by taking a contrapositive, we obtain

for any call sequence π which follows $\mathcal{P}$,
M_π has no infinite descents implies that π is not infinite.

This is the sentence from which we will obtain our definition. Formally a call sequence which follows $\mathcal{P}$ is a function $\pi : \mathbb{N} \longrightarrow \{\tau \mid \tau \text{ is a call in } \mathcal{P}\} \cup \{\emptyset\}$ such that

- if $\pi(n) = \emptyset$ for some $n \in \mathbb{N}$, then $\forall m > n(\pi(m) = \emptyset)$;
- if $\pi(n+1) = \tau$, then τ is a call which appears in the definition corresponding to the call $\pi(n)$.

Observe that π is infinite in this notation means that $\forall n(\pi(n) \neq \emptyset)$. In order to keep the notation of [14] for any natural number n we denote $\tau_n = \pi(n)$.

We introduce two binary relations, π^+ on $\mathbb{N}$ and R_π on $\mathbb{N} \times \mathrm{Var}$. Then we translate "$M_\pi$ has no infinite descents" with "R_π is inductively well-founded" and "π is not infinite" with "π^+ is inductively well-founded".

Let $\mathcal{P}$ be a program and let π be a call sequence which follows $\mathcal{P}$. We define a binary relation π^+ on $\mathbb{N}$ by:

$$m\pi^+ n \iff m > n \wedge \tau_m \neq \emptyset.$$

Observe that if π is infinite then $m\pi^+ n$ holds if and only if $m > n$, while if l is the minimum number such that $\pi(l) = \emptyset$ then $m\pi^+ n$ holds if and only if $l > m > n$.

Now, we define a binary relation R_π on $\mathbb{N} \times \mathrm{Var}$. Here $(m,y)R_\pi(n,x)$ holds if and only if y becomes strictly smaller than x when we step from τ_n to τ_{m-1} along the call sequence π.

▶ **Definition 16.** Given a sequence π that follows $\mathcal{P}$, R_π is defined as:

$$(m,y)R_\pi(n,x) \iff m\pi^+ n \wedge x \xrightarrow{\downarrow} y \in G_{\tau_n}; \ldots G_{\tau_{m-1}},$$

where G_τ is the size-change graph associated to τ.

From R_π and π^+ we define SCT*.

▶ **Definition 17** (SCT* program). $\mathcal{P}$ is SCT* if and only if for any call sequence π which follows $\mathcal{P}$: R_π is (inductively) well-founded implies that π^+ is (inductively) well-founded.

We highlighted the use of the inductive definition of well-foundedness instead of the classical one, since as seen it is crucial in order to give an intuitionistic proof of the SCT Theorem. However for short, from now on we will write well-foundedness instead of inductive well-foundedness.

4 Proving SCT* Theorem

The goal of this section is to intuitionistically prove that the SCT Theorem by Lee et al. holds providing we use SCT* instead of SCT. The SCT Theorem states that a program $\mathcal{P}$ is SCT if and only if for any idempotent $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ there exists a variable x in the source of G such that $x \overset{\downarrow}{\to} x \in G$. Recall that in the classical proof of the SCT Theorem the main ingredient is Ramsey's Theorem for pairs. In the classical proof the authors suppose by contradiction that there exists an infinite call sequence π which follows $\mathcal{P}$. Then they define a coloring $h : [\mathbb{N}]^2 \to \mathsf{cl}(\mathcal{G}_\mathcal{P})$ which associates to any pair of natural numbers $n < m$ the size-change graph which corresponds to $G_{\tau_n}; \ldots; G_{\tau_{m-1}}$. Since $\mathsf{cl}(\mathcal{G}_\mathcal{P})$ if finite and by applying Ramsey's Theorem for pairs they obtain an infinite homogeneous chain and therefore a contradiction.

We will prove that also in this case we can use the H-closure Theorem instead of Ramsey's Theorem for pairs. In order to do that we need to introduce a binary relation π^+_G for any $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ and for any call sequence π which follows $\mathcal{P}$. We have that $m\pi^+_G n$ holds if and only if the size-change graph associated to $\tau_n, \ldots, \tau_{m-1}$ is G.

▶ **Definition 18.** Let π be a call sequence which follows $\mathcal{P}$ and let $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$. Define $\pi^+_G \subseteq \mathbb{N}^2$ as:

$$m\pi^+_G n \iff m\pi^+ n \wedge G_{\tau_n}; \ldots; G_{\tau_{m-1}} = G.$$

Observe that π^+_G is decidable since G_{τ_i} is finite and π^+ is decidable. Moreover, as remarked in Subsection 3.3, if R is decidable then $H(R)$ is.

▶ **Lemma 19.** *Let π be a call sequence which follows $\mathcal{P}$ and let $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$. Then both π^+_G and $H(\pi^+_G)$ are decidable.*

By applying the H-closure Theorem to the relations π^+_G, for $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$, we can intuitionistically prove the SCT* Theorem.

▶ **Theorem 20** (SCT* Theorem). *Every idempotent graph in $\mathsf{cl}(\mathcal{G}_\mathcal{P})$ has an edge $x \overset{\downarrow}{\to} x$ if and only if $\mathcal{P}$ is* SCT*.

Proof. "⇒": Assume that any idempotent graph in $\mathsf{cl}(\mathcal{G}_\mathcal{P})$ has an edge $x \overset{\downarrow}{\to} x$. Let π be a call sequence which follows $\mathcal{P}$ such that R_π is well-founded. We want to prove that π^+ is well-founded.

▶ **Claim.** *For any G in $\mathsf{cl}(\mathcal{G}_\mathcal{P})$, π^+_G is H-well-founded.*

Proof of the Claim. Since "G is idempotent" is a decidable statement we can consider two cases.

- If G is not idempotent, then each $L \in H(\pi_G^+)$ has length at most 2. Otherwise assume that $\langle n, m, l\rangle \in H(\pi_G^+)$ for some $n < m < l$. By definition, this would imply that $m\pi_G^+ n$, $l\pi_G^+ m$ and $l\pi_G^+ n$, therefore we would have

  $$G; G = G_{\tau_n}; \ldots; G_{\tau_{m-1}}; G_{\tau_m}; \ldots; G_{\tau_{l-1}} = G.$$

  This means that G is idempotent. Contradiction. Hence we have $\neg\exists n, m, l(\langle n, m, l\rangle \in H(\pi_G^+))$, and so $\forall n, m, l(\langle n, m, l\rangle \notin H(\pi_G^+))$ follows by Lemma 19. Hence π_G^+ is H-well-founded.
- If G is idempotent, then there exists $x \xrightarrow{\downarrow} x \in G$. Define the following binary relation

  $$U_x = \{(n, (n, x)) \mid n \in \mathbb{N}\}.$$

  Then U_x is a simulation of π_G^+ in R_π. In fact assume that

  $$m\pi_G^+ n \ \wedge \ nU_x(n, x),$$

  Therefore, since $x \xrightarrow{\downarrow} x \in G = G_{\tau_n}; \ldots; G_{\tau_{m-1}}$, we have

  $$mU_x(m, x) \wedge (m, x)R_\pi(n, x).$$

  Since by hypothesis R_π is well-founded, then by Proposition 5 also π_G^+ is well-founded. By Proposition 8 it is H-well-founded. ◀

Now observe that

$$\pi^+ = \bigcup \{\pi_G^+ \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\},$$

since every $G_{\tau_n}; \ldots; G_{\tau_{m-1}}$ equates some $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ by definition of $\mathsf{cl}(\mathcal{G}_\mathcal{P})$. Hence by applying both the H-closure Theorem (Theorem 9) and finiteness of $\mathsf{cl}(\mathcal{G}_\mathcal{P})$ we obtain also π^+ is H-well-founded. Moreover it is transitive by definition, then it is well-founded by Proposition 8 and we are done.

"$\Leftarrow$": Suppose that $\mathcal{P}$ is SCT* and let G_τ be an idempotent size-change graph. By idempotency if we define the call sequence π such that $\pi(n) = \tau$ for any $n \in \mathbb{N}$, then it is an infinite call sequence which follows $\mathcal{P}$. In particular π^+ is not well-founded. Since $\mathcal{P}$ is SCT*, R_π is not well-founded. In this case, we may observe that:

$$(m, y)R_\pi(n, x) \iff x \xrightarrow{\downarrow} y \in G_{\tau_n}; \ldots; G_{\tau_{m-1}} = G_\tau; \ldots; G_\tau = G_\tau$$

Then $(m, y)R_\pi(n, x) \iff x \xrightarrow{\downarrow} y \in G_\tau$. Define

$$y\tilde{R}_\pi x \iff x \xrightarrow{\downarrow} y \in G_\tau,$$

Observe that if $\tilde{R}_\pi$ is well-founded, then also R_π is. We can prove it by using the simulation

$$U = \{((n, x), x) \mid n \in \mathbb{N}, x \text{ a variable in the source of } G_\tau\}.$$

Hence $\tilde{R}_\pi$ is not well-founded. Since $\tilde{R}_\pi$ is not well-founded and it is finite, thanks to Proposition 6, it has a cycle for some variable z:

$$z = z_n \tilde{R}_\pi z_{n-1} \tilde{R}_\pi \ldots \tilde{R}_\pi z_0 = z.$$

Moreover $\tilde{R}_\pi$ is transitive: in fact if $x \xrightarrow{\downarrow} y \in G_\tau$ and $y \xrightarrow{\downarrow} z \in G_\tau$, then $x \xrightarrow{\downarrow} z \in G_\tau; G_\tau = G_\tau$. Since $\tilde{R}_\pi$ is transitive, hence $z\tilde{R}_\pi z$. Therefore

$$z \xrightarrow{\downarrow} z \in G_\tau.$$

We have proved that if G_τ is idempotent, then $z \xrightarrow{\downarrow} z \in G_\tau$. ◀

By comparing the classical proofs of the termination theorems, the version of Ramsey's Theorem for pairs used in the proof of the Termination Theorem is weaker then the one used in the proof of the SCT Theorem. In fact in order to prove the Termination Theorem it is sufficient to have an infinite homogeneous chain (i.e. an ordered set $\{x_i \mid i \in \mathbb{N}\}$ such that each pair of consecutive elements has the same color) instead of an infinite homogeneous set. This result which is an infinite version of Erdős–Szekeres's Theorem [10] is called also Weak Ramsey's Theorem for pairs and it is strictly weaker than Ramsey's Theorem for pairs in two colors in RCA_0 [7]. We can stress this difference also in the intuitionistic proofs. In fact the proof of the Termination Theorem in [6] uses:

if $R_0, \dots, R_{n-1}$ *are well-founded then* $\bigcup\{R_i \mid i < n\}$ *is* H*-well-founded,*

where the hypothesis is stronger than in the H-closure Theorem, by Proposition 8. On the other hand the proof of the SCT* Theorem above uses the whole H-closure Theorem.

Let us conclude this section with an example of an SCT* program.

▶ **Example 21.** Let us consider the following functional program, where $*$ denotes any value.

$$\begin{aligned} g(x,y,\mathrm{temp},\mathrm{exp},z) :=& \text{if } (z=0) \quad 0 \\ & \text{else if } (z=1) \quad \mathrm{temp} \\ & \text{else } \tau_0 : g(*,*,\mathrm{temp}+\mathrm{exp},\mathrm{exp},z-1) \\ f(x,y,\mathrm{temp},\mathrm{exp},z) :=& \text{if } (y=0) \quad 1 \\ & \text{else if } (y=1) \quad \mathrm{exp} \\ & \text{else } \tau_1 : f(x,y-1,*,g(x,y,0,\mathrm{exp},x)),*) \end{aligned}$$

As in Example 3, $f(x,y,0,1,z)$ computes x^y. The idempotent graphs in $\mathsf{cl}(\mathcal{G})$ are $G_{\tau_0} : g \to g$ and $G_{\tau_1} : f \to f$ (since the source and the target of the other size-change graphs are different). G_{τ_0} is composed of $z \xrightarrow{\downarrow} z$ and $\mathrm{exp} \xrightarrow{\Downarrow} \mathrm{exp}$, while G_{τ_1} is composed of $y \xrightarrow{\downarrow} y$ and $x \xrightarrow{\Downarrow} x$. Hence this program is SCT* since it satisfies the condition of the SCT* Theorem.

5 Tail-recursive SCT* programs compute exactly primitive recursive functions

In this section we compare size-change termination and transition invariant termination. As Heizmann, Jones and Podelski did, we restrict the domain of the programs we consider in order to match transition invariants termination and SCT*. In fact SCT (and so SCT*) is defined for functional programs, while Podelski and Rybalchenko's termination is defined for transition-based programs. As they did from now on we consider just tail-recursive functional programs (where all functions use the same variables), for which there exists a direct transition-based translation into while-if programs. We refer to [13] for details. The reader has to keep in mind that along all this section we have at the same time a functional program, which we denote by $\mathcal{P}$, and its translation as a transition-based program $\mathcal{R}_\mathcal{P}$. The only property we use of the translation $\mathcal{R}_\mathcal{P}$ is that $\mathcal{R}_\mathcal{P}$ consists of: while, if, a program counter and the values of the variables of $\mathcal{P}$. If $\mathcal{P}$ were recursive but not tail-recursive, $\mathcal{R}_\mathcal{P}$ should include also a stack, but we explicitly assume that this is not the case. We will derive a characterization of $\mathcal{P}$ from a characterization of $\mathcal{R}_\mathcal{P}$.

The goal of this section is to prove, by using the result obtained in [4], that the functional programs which are tail-recursive and are SCT* compute exactly the primitive recursive functions. Ben-Amram in [3] already proved that the tail-recursive SCT programs compute

primitive recursive functions, however we present a completely different proof which uses an analysis of the intuitionistic proof of the Termination Theorem, in the case of a transition invariant of height ω. In fact we intuitionistically prove that if a program $\mathcal{P}$ is SCT* then $\mathcal{R}_\mathcal{P}$ has a transition invariant of height ω.

First of all we recall some definitions and results useful to compare size-change termination and Podelski and Rybalchenko's termination. Each state in the transition-based program $\mathcal{R}_\mathcal{P}$ which corresponds to the tail-recursive functional program $\mathcal{P}$ is a tuple s composed of the location $s(\mathrm{pc})$ of the program instruction and a value $s(x)$ for any variable x. We define a relation $\Phi(G)$ on states saying that whenever G includes a decreasing edge $x \xrightarrow{\downarrow} y$ then the value of y in the second state is smaller than the value of x in the first state, and similarly for any weakly-decreasing edge.

▶ **Definition 22** (Transition relation of a size-change graph, Definition 26 [13])**.** Given a size-change graph $G : f \to g$, define the binary relation over states $\Phi(G) \subseteq S \times S$ as follows: $s'\Phi(G)s$ if and only if $s(\mathrm{pc}) = f$, $s'(\mathrm{pc}) = g$ and

$$\bigwedge \left\{ s(z_i) \geq s'(z_j) \mid (z_i \xrightarrow{\Downarrow} z_j) \in G \right\} \wedge \bigwedge \left\{ s(z_i) > s'(z_j) \mid (z_i \xrightarrow{\downarrow} z_j) \in G \right\}.$$

The transition relation ρ_τ associated to the transition

$$f(x_0, \ldots, x_{n-1}) = \ldots \tau : g(e_0, \ldots, e_{n-1}), \ldots,$$

is defined as follows:

$$\rho_\tau = \left\{ ((f, \mathbf{v}), (g, \mathbf{u})) \mid (f, \mathbf{v}) \xrightarrow{\tau} (g, \mathbf{u}) \right\}.$$

Observe that if G_τ is the size-change graph assigned to the call τ of program $\mathcal{P}$, G_τ is safe for τ if and only if the inclusion $\rho_\tau \subseteq \Phi(G_\tau)$ holds.

▶ **Lemma 23** (Lemma 29 [13])**.** *The composition of the two size-change graphs $G_1 : f \to g$ and $G_2 : g \to h$ overapproximates the composition of the relations they define, i.e.*

$$\Phi(G_1) \circ \Phi(G_2) \subseteq \Phi(G_1; G_2).$$

The authors of [13] proved the following lemma about the connection between G and $\Phi(G)$.

▶ **Lemma 24** (Lemma 31 [13])**.** *Let G be a size-change graph such that source and target of G coincide. If G has an edge of form $x \xrightarrow{\downarrow} x$ then the relation $\Phi(G)$ is well-founded.*

They also noticed that the vice versa does not hold in the general case, however we prove that if G is idempotent this equivalence holds.

▶ **Lemma 25.** *Let G be an idempotent size-change graph. Then G has an edge of form $x \xrightarrow{\downarrow} x$ if and only if the relation $\Phi(G)$ is well-founded.*

Proof.
"⇒": It follows from Lemma 24.
"⇐": Observe that "there exists a variable x in the source of G, $x \xrightarrow{\downarrow} x \in G$" is a decidable statement, since G is finite. Therefore either G has some edge of the form $x \xrightarrow{\downarrow} x$, or G has no edge of this form. In the first case we are done, in the second one we will prove a contradiction, that $\Phi(G)$ is not well-founded. This argument intuitionistically shows the

thesis, because from a contradiction we may derive everything. Let V be the set of the variables in the source of G. Define the following preorder on V:

$$x \precsim y \iff x = y \ \vee \ y \xrightarrow{\downarrow} x \ \vee \ y \xrightarrow{\Downarrow} x.$$

It is reflexive by definition and it is transitive since G is idempotent: if $y \xrightarrow{r} x \in G$ and $z \xrightarrow{r'} y \in G$ then $z \xrightarrow{r''} x \in G$ for some $r'' \in \{\downarrow, \Downarrow\}$.

Let us consider the quotient X of V with respect to the equivalence relation

$$x \sim y \iff x \precsim y \ \wedge \ y \precsim x.$$

In X, $\precsim$ is an order since it is also antisymmetric. Moreover X is finite then, by Proposition 6, $\precsim$ is well-founded on X. Hence for any equivalence class $[x] \in X$ we can define $h([x])$ to be the height of $[x]$ with respect to $\precsim$ in X: i.e. the number of elements $[y] \in X$ such that $[y] \precsim [x]$.

Then we can define a state s as follows: for any $x \in V$, $s(x) = h([x])$. We claim that $s\Phi(G)s$. In fact

- if $y \xrightarrow{\Downarrow} x \in G$ then $x \precsim y$ and we have two possibilities: if $[x] = [y]$ in X then $s(x) = s(y)$, otherwise $h([x]) < h([y])$ and so $s(x) < s(y)$;
- if $y \xrightarrow{\downarrow} x \in G$ then $x \precsim y$. Moreover $y \precsim x$ is false, otherwise by case analysis from $y \xrightarrow{\downarrow} x \in G$ and $x = y \vee x \xrightarrow{\downarrow} y \in G \vee x \xrightarrow{\Downarrow} y \in G$ we deduce $x \xrightarrow{\downarrow} x \in G$, contradicting the hypothesis. Hence $h([x]) < h([y])$ and so $s(x) < s(y)$.

Then $s\Phi(G)s$ and so $\Phi(G)$ is ill-founded. Contradiction. ◀

Thanks to the SCT* Theorem and the lemma above, we may observe that if $\mathcal{P}$ is tail-recursive, $\mathcal{P}$ is SCT* if and only if for every $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ idempotent $\Phi(G)$ is well-founded.

Our next goal is to prove that any tail-recursive program which is SCT* computes a primitive recursive function. In order to do that we modify the proof by Heizmann, Jones and Podelski of "$\bigcup \{\Phi(G) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}$ is a transition invariant". We will prove that it is a transition invariant of height ω. In order to do this we need the following lemmas.

▶ **Lemma 26.** *Every finite semigroup G has an idempotent element.*

Proof. Let $x \in G$ and consider the following chain

$$x \mapsto x^2 \mapsto (x^2)^2 = x^4 \mapsto \ldots \mapsto x^n \mapsto (x^n)^2 \mapsto \ldots.$$

Since G is finite, there exists y in the previous chain such that $y^k = y$, for some $k \geq 2$. Put $z = y^{k-1}$, then

$$z \cdot z = y^{k-1} \cdot y^{k-1} = y^k \cdot y^{k-2} = y \cdot y^{k-2} = y^{k-1} = z.$$ ◀

Even if the previous definitions and results can be stated for any functional program, we highlight now that we need tail-recursive functional programs in order to have the translation $\mathcal{R}_\mathcal{P}$ of $\mathcal{P}$. In this case each state of $\mathcal{R}_\mathcal{P}$ is composed of the location of the program and the values in $\mathbb{N}$ of the variables.

▶ **Lemma 27.** *Let G be a size-change graph. Let k be a positive natural number. If G^k is such that $x \xrightarrow{\downarrow} x$ for some x then $\Phi(G)$ has height ω.*

Proof. Assume that $x \overset{\downarrow}{\to} x$ in G^k. We distinguish the cases $k = 1$ and $k \geq 2$.

If $k = 1$, let $f : \mathrm{dom}(\Phi(G)) \to \mathbb{N}$ be such that $f(s) = s(x)$. Since If $s'\Phi(G)s$, then by $x \overset{\downarrow}{\to} x \in G$ we deduce $s'(x) < s(x)$, hence $f(s') < f(s)$. Thus f is a weight function for $\Phi(G)$.

Assume now that $k \geq 2$, since $x \overset{\downarrow}{\to} x$ in G^k, then there exist $y_0, \ldots, y_{k-2}$ such that

$$x \to y_0 \to \cdots \to y_{k-2} \to x$$

where at least one of these arrows is strictly decreasing. Then define a function $f : \mathrm{dom}(\Phi(G)) \to \mathbb{N}$ by

$$f(s) = \sum_{i=0}^{k-2} s(y_i) + s(x).$$

Hence if $s'\Phi(G)s$ then each of the $s'(y_i)$ and $s'(x)$ is less or equal to the ones of s. Moreover one of these is strictly less, since at least one of the edges of G is strictly decreasing. So $f(s') < f(s)$ and this means that f witnesses that $\Phi(G)$ has height ω. ◀

By using the results above we can modify the Theorem Idempotence and well-foundedness [13, Theorem 32] which states that if for any $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ idempotent $\Phi(G)$ is well-founded, then $\Phi(G)$ is well-founded for any $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$.

▶ **Theorem 28.** *If*

$$\forall G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})(G; G = G \implies \Phi(G) \textit{ is well-founded})$$

then $\Phi(G)$ has height ω for every graph in $\mathsf{cl}(\mathcal{G}_\mathcal{P})$.

Proof. Let $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ be a size-change graph. There are two possibilities. On the one hand, if the source and target of G do not coincide, then $\Phi(G) \circ \Phi(G) = \emptyset$, therefore $\Phi(G)$ has height ω. On the other hand, assume that source and target of G coincide. Then G^n is defined for any $n \in \mathbb{N}$. Since the semigroup $(\{G^n \mid n > 0\}, ;)$ is finite, by Lemma 26 it has an idempotent element G^k. Since $\Phi(G^k)$ is well-founded by hypothesis we obtain, by applying Lemma 25, that $x \overset{\downarrow}{\to} x \in G^k$ for some x. By Lemma 27 $\Phi(G)$ has height ω and we are done. ◀

▶ **Corollary 29** (Corollary 33 [13])**.** *If the program $\mathcal{P}$ is size-change terminating for a set of size-change graphs $\mathcal{G}_\mathcal{P}$ that is a safe description of $\mathcal{P}$, then the relation defined by its closure $\mathsf{cl}(\mathcal{G}_\mathcal{P})$*

$$\bigcup \{\Phi(G) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}$$

is a disjunctively well-founded transition invariant for $\mathcal{R}_\mathcal{P}$.

Therefore, thanks to Theorem 28, all $\Phi(G)$ have height ω: by definition, the transition invariant has height ω. In [4] and [11] it is proved that if $\mathcal{R}$ has a transition invariant of height ω then it computes a primitive recursive function.

We apply this result to the while-if program $\mathcal{R}_\mathcal{P}$ which translates the tail-recursive program P. We obtain:

▶ **Proposition 30.** *Each tail-recursive program which is* SCT* *is primitive recursive.*

This result was already proved by Ben-Amram in [3] using the classical definition of SCT. He proved that in general SCT programs compute multiple recursive function. As a corollary, by observing that if you do not use nested recursive calls a multiple recursive function is primitive recursive, he obtained that any tail-recursive SCT program computes a primitive recursive function.

By using our proof we can easily obtain a bound whose class is given by the number of relations of the transition invariant. In fact the weight function provided in Lemma 27 is in $\mathcal{F}_1$, since $f(s) < |s| \cdot F_0(\max(s))$ and for any program $|s|$ is fixed. Therefore by applying the bound provided in [11] we have that if the transition relation of $\mathcal{R}_\mathcal{P}$ is the graph of a function in $\mathcal{F}_n$, there is a bound in $\mathcal{F}_{k+n-1}$ where k is the number of the relations which compose the transition invariant whose weight functions are in $\mathcal{F}_1$. Therefore by Corollary 29, we can conclude that if the transition relation is in $\mathcal{F}_2$, the function is in $\mathcal{F}_{|\mathsf{cl}(\mathcal{G}_\mathcal{P})|+1}$. Unfortunately $\mathsf{cl}(\mathcal{G}_\mathcal{P})$ is exponential in $\mathcal{G}_\mathcal{P}$, so this bound is huge. We have also another bound on the number of variables. In fact in the proof of Theorem 28 we saw that for any G there exists $k > 0$ such that G^k is idempotent. Let us consider the minimum such k. Then, by following the proof of Lemma 27 there exists either a $x \xrightarrow{\downarrow} x$ for some x or a chain

$$x \to y_0 \to \cdots \to y_{k-2} \to x$$

for some variables, where at least one arrow is strict. Observe that all the variables in the chain are different: there is not a path which connect some y_i to itself, by minimality of k. The weight function we built for $\Phi(G)$ is given by the sum of the values which corresponds to these variables. This means that if we have n-many variables, the number of possible weight functions f of this kind is

$$\sum_{i=k}^{n} \binom{n}{k} = 2^n - 1.$$

Since if R and R' have the same weight function, then also $R \cup R'$ have this weight function, we can merge the relations in the transition invariant found in such a way their number is less or equal to the number of the possible weight functions. Unfortunately also this bound is exponential (in the number of variables), so it is huge too.

▶ **Example 31.** The program considered in Example 21 is tail-recursive. Observe that

$$\begin{aligned}
\Phi(G_{\tau_0}) &= \{s'\Phi(G)s \mid s'(\mathrm{pc}) = s(\mathrm{pc}) = g \ \wedge \ s'(z) < s(z) \ \wedge \ s'(\mathrm{exp}) \leq s'(\mathrm{exp})\}\,;\\
\Phi(G_{\tau_1}) &= \{s'\Phi(G)s \mid s'(\mathrm{pc}) = s(\mathrm{pc}) = f \ \wedge \ s'(y) < s(y) \ \wedge \ s'(x) \leq s'(x)\}\,;\\
\Phi(G_{\tau_2}) &= \{s'\Phi(G)s \mid s'(\mathrm{pc}) = f \ \wedge \ s(\mathrm{pc}) = g\}\,.
\end{aligned}$$

Then $\Phi(G_{\tau_0}) \cup \Phi(G_{\tau_1}) \cup \Phi(G_{\tau_2})$ is already a transition invariant of height ω for the transition-based program which corresponds to it (Example 3 where $l = g$ and $l' = f$). Trivially, also $\bigcup\{\Phi(G) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}$ is a transition invariant and the function computed is primitive recursive.

Furthermore we can observe that each primitive recursive function has a tail-recursive implementation which is SCT*.

▶ **Proposition 32.** *Each primitive recursive function has an implementation which is* SCT*.

Proof. By induction on the primitive recursive functions.

- For the constant function, successor function and the projection function it is trivial since we can write tail-recursive first order functional programs which have no idempotent size-change graphs.
- Assume that the primitive recursive functions $g_0, \dots, g_{n-1}$ and f have an implementation which is SCT*. By using them it is straightforward to show that the standard program which computes their composition

$$h(x_0, \dots, x_{k-1}) = f(g_0(x_0, \dots, x_{k-1}), \dots, g_{n-1}(x_0, \dots, x_{k-1}))$$

  is SCT*. In fact each idempotent size-change graph corresponds to some call in the definitions either of g_i for some $i < k$ or of f.
- Assume that the primitive recursive functions f and g have a SCT* program which computes it. Then, by using these programs we can define a standard tail-recursive SCT* program which computes

$$h(x_0, \dots, x_{k-1}, y) = \begin{cases} f(x_0, \dots, x_{k-1}) & \text{if } y = 0 \\ g(h(x_0, \dots, x_{k-1}, y-1), y) & \text{otherwise.} \end{cases}$$

  As observed in the previous point each size-change graph which corresponds to some call either in f or in g, has the desired property. There is only one new size-change graph $G \in \mathsf{cl}(\mathcal{G}_{\mathcal{P}})$ derived from the definition of h. Since $y \overset{\downarrow}{\to} y \in G$, we are done. ◀

6 Transition Invariants Termination as a property of size-change graphs

In the previous section we proved that if a tail-recursive program is SCT* then it has a transition invariant of height ω. In this section we will see a statement on functional programs strictly weaker than SCT* which is equivalent to the definition of termination by Podelski and Rybalchenko, so it is equivalent to have a transition invariant of general height.

Thanks to Lemma 25, we saw that if $\mathcal{P}$ is tail-recursive, $\mathcal{P}$ is SCT* if and only if for every $G \in \mathsf{cl}(\mathcal{G}_{\mathcal{P}})$ idempotent $\Phi(G)$ is well-founded. Recall that Podelski and Rybalchenko analyse the termination of while-if programs, and in order to have a simple relationship with while-if program we restrict the functional programs to be tail-recursive. Here we prove that a tail-recursive program $\mathcal{P}$ has a disjunctively well-founded transition invariant if and only if for any $G \in \mathsf{cl}(\mathcal{G}_{\mathcal{P}})$ idempotent $\Phi(G) \cap R^+$ is well-founded, where R is the transition relation of $\mathcal{R}_{\mathcal{P}}$. The proof of "for any G idempotent $\Phi(G) \cap R^+$ is well-founded implies termination" follows, with some little changes, the proof of Corollary 29 studied in [13, Corollary 33].

The first step is to intuitionistically prove another version of the Theorem Idempotence and well-foundedness [13, Theorem 32]. In order to do that we need the following lemma.

▶ **Lemma 33.** *Let R be a binary relation on I and $k \in \mathbb{N}$ and T a transitive binary relation. If $R^k \cap T$ is well-founded then $R \cap T$ is well-founded.*

Proof. Let $k = 2$. Induction on x with respect to R^2. Assume that

$$\forall z(z(R^2 \cap T)x \implies z \text{ is } (R \cap T)\text{-well-founded }).$$

By two applications of point 1 of Proposition 5,

$$\begin{aligned} x \text{ is } (R \cap T)\text{-well-founded} &\iff \forall y(y(R \cap T)x \implies y \text{ is } (R \cap T)\text{-well-founded}) \\ &\iff \forall y(y(R \cap T)x \implies (\forall z(z(R \cap T)y \implies z \text{ is } (R \cap T)\text{-well-founded}))). \end{aligned}$$

Observe that since $z(R \cap T)y$ and $y(R \cap T)x$ then $z(R^2 \cap T)x$. This implies by inductive hypothesis that z is $(R \cap T)$-well-founded. So for every $x \in I$, x is $(R \cap T)$-well-founded.

- The idea of the proof for $k > 2$ is to prove it by induction on x with respect to R^k and to repeat the same argument providing in the case above, by using k-many steps following point 1 of Proposition 5 in order to get $z(R^k \cap T)x$. By applying the inductive hypothesis we will obtain our thesis. ◀

▶ **Theorem 34.** *If*

$$\forall G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})(G; G = G \implies \Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \text{ well-founded })$$

then $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ *is well-founded for every graph in* $\mathsf{cl}(\mathcal{G}_\mathcal{P})$.

Proof. Let $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ be a size-change graph. We have two cases. On the one hand, if the source and target of G do not coincide, then $\Phi(G) \circ \Phi(G) = \emptyset$, therefore $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded. On the other hand, assume that source and target of G coincide. In this case G^n is defined for all $n \in \mathbb{N}$. Since the semigroup $(\{G^n \mid n \in \mathbb{N}\}, ;)$ is finite, by Lemma 26 it has an idempotent element G^k. By Lemma 23 the inclusion $\Phi(G)^k \subseteq \Phi(G^k)$ holds. Then

$$\Phi(G)^k \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \subseteq \Phi(G^k) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$$

By hypothesis, since G^k is idempotent we have: $\Phi(G^k) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded. Then $\Phi(G)^k \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded by Lemma 23 and therefore by Lemma 33 also $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded. ◀

Therefore we obtain the corresponding version of Corollary 29.

▶ **Corollary 35.** *Let $\mathcal{P}$ be a program and let $\mathcal{G}_\mathcal{P}$ be a set of size-change graphs that is a safe description of $\mathcal{P}$. If for every $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ idempotent, $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded, then the relation defined by its closure $\mathsf{cl}(\mathcal{G}_\mathcal{P})$*

$$\bigcup \{\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}$$

is a disjunctively well-founded transition invariant for $\mathcal{R}_\mathcal{P}$.

Proof. Thanks to the proof of Corollary 29 in [13, Corollary 33]

$$R^+ \subseteq \bigcup \{\Phi(G) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}.$$

Then

$$R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \subseteq \bigcup \{\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}.$$

Moreover, by hypothesis and thanks to Lemma 34 the relation $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded. Hence $\bigcup \{\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \mid G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})\}$ is a disjunctively well-founded transition invariant for $\mathcal{R}_\mathcal{P}$. ◀

Finally we prove the equivalence, the other implication is trivial.

▶ **Theorem 36.** *Given a program $\mathcal{P}$ the followings are equivalent:*

1. *for every $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$ idempotent, $\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded;*
2. *there is a disjunctively well-founded transition invariant for $\mathcal{R}_\mathcal{P}$.*

Proof. "⇑": If 2 holds, then $R^+ \cap (\mathrm{Acc} \times \mathrm{Acc})$ is well-founded. Then for every $G \in \mathsf{cl}(\mathcal{G}_\mathcal{P})$

$$\Phi(G) \cap R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}) \subseteq R^+ \cap (\mathrm{Acc} \times \mathrm{Acc}),$$

is well-founded.

"⇓": If 1 holds, then by Corollary 35 we obtain the thesis. ◀

To conclude observe that the condition (1) in the previous theorem is strictly weaker than being SCT. To put otherwise: using a transition invariant, we may prove terminating some programs $\mathcal{R}_\mathcal{P}$ which are while-if translation of non-SCT tail-recursive programs $\mathcal{P}$. The following basic example explains why.

▶ **Example 37.** Let us consider the following functional program:

$$\begin{aligned} f(x,y) :=& \text{if } (x > y) \quad x \\ & \text{else } \tau : f(x+1, y). \end{aligned}$$

It is not SCT since G_τ is idempotent and has no decreasing edges. In particular $\Phi(G)$ is not well-founded. However $\Phi(G) \cap R^+$ is and therefore this program satisfies the condition (1) of Theorem 36, therefore in particular it is terminating.

7 Conclusions

In this work we presented an intuitionistic proof of the SCT* Theorem. This is not the first intuitionistic proof of the SCT Theorem. Vytiniotis, Coquand and Wahlsteldt in [20] intuitionistically proved it by using Almost-Full relations. A binary relation R over a set S is almost-full if the set of finite sequences $x_0, x_1, \ldots, x_n$ on S, such that for no $i < j \leq n$ $x_i R x_j$ holds, is inductively well-founded. Classically, the set of almost-full relations R is the set of relations such that the complement of the inverse of R is H-well-founded. However, we need De Morgan's Law to prove this equivalence. Therefore it is not evident whether the H-closure Theorem may be intuitionistically derived from the Almost-full Theorem, or the other way round. The proof of the SCT Theorem in [20] uses the following facts:

- almost-full relations are closed under finite intersections;
- if R and T are two binary relations such that $T \cap R^{-1} = \emptyset$ and R is almost-full, then T is well-founded.

In our intuitionistic proof, instead, we use H-well-founded relations and:

- H-well-founded relations are closed under finite unions;
- if a binary relation R is H-well-founded and transitive then it is well-founded.

In [6] we showed that we may provide an intuitionistic proof of the Termination Theorem by replacing the use of Ramsey's Theorem for pairs with the use of the H-closure Theorem. In this paper we did the same for the SCT Theorem. Since Ramsey's Theorem for pairs is used in many branches of mathematics, in future works we hope to apply this method to other classical results based on it, in order to obtain intuitionistic proofs.

We proved that the functions which are computed by a tail-recursive SCT* program are exactly the primitive recursive functions. This result fits in with the one by Ben-Amram [3]: he proved that the SCT programs compute primitive recursive functions. More in details, for any tail-recursive SCT program we provided some primitive recursive bound to the number of computation steps given an input. However as discussed in Section 5 the bound obtained in this way is large. An open question is whether we may extract from the intuitionistic proof of the SCT* Theorem a bound stricter than this one.

Acknowledgements. I would like to thank Stefano Berardi for his careful readings, remarks and corrections, and the anonymous referees for their very helpful suggestions.

References

1 Peter Aczel. *Handbook of Mathematical Logic*, volume 90 of *Studies in Logic and the Foundations of Mathematics*, chapter An introduction to inductive definitions, pages 739–782. Elsevier, 1977.

2 Thorsten Altenkirch. A formalization of the strong normalization proof for system F in LEGO. In *TLCA*, pages 13–28, 1993.

3 Amir M. Ben-Amram. General size-change termination and lexicographic descent. In Torben Mogensen, David Schmidt, and I. Hal Sudborough, editors, *The Essence of Computation: Complexity, Analysis, Transformation. Essays Dedicated to Neil D. Jones*, volume 2566 of *LNCS*, pages 3–17. Springer-Verlag, 2002.

4 Stefano Berardi, Paulo Oliva, and Silvia Steila. Proving termination with transition invariants of height omega. *CoRR*, abs/1407.4692, 2014.

5 Stefano Berardi and Silvia Steila. Ramsey theorem for pairs as a classical principle in intuitionistic arithmetic. In *TYPES*, pages 64–83, 2013.

6 Stefano Berardi and Silvia Steila. Ramsey theorem as an intuitionistic property of well founded relations. In *RTA-TLCA*, pages 93–107, 2014.

7 Stefano Berardi, Silvia Steila, and Keita Yokoyama. Notes on H-closure. In preparation.

8 Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Abstraction refinement for termination. In *SAS*, pages 87–101, 2005.

9 Thierry Coquand. A direct proof of Ramsey's Theorem. Author's website, revised in 2011, 1994.

10 Paul Erdős and George Szekeres. A combinatorial problem in geometry. *Compositio Mathematica*, 2:463–470, 1935.

11 Diego Figueira, Santiago Figueira, Sylvain Schmitz, and Philippe Schnoebelen. Ackermannian and primitive-recursive bounds with Dickson's Lemma. In *LICS*, pages 269–278. IEEE Press, 2011.

12 Alfons Geser. Relative termination. PhD thesis, Universitat Passau, 1990.

13 Matthias Heizmann, Neil D. Jones, and Andreas Podelski. Size-change termination and transition invariants. In *SAS*, pages 22–50, 2010.

14 Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *POPL*, pages 81–92, 2001.

15 M.H. Löb and S.S. Wainer. Hierarchies of number-theoretic functions. i. *Arch. Math. Logic*, 13(1-2):39–51, 1970.

16 David Park. Concurrency and automata on infinite sequences. In *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981.

17 Lawrence C. Paulson. Constructing recursion operators in intuitionistic type theory. *J. of Symb. Comp.*, 2(4):325–355, 1986.

18 Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *LICS*, pages 32–41, 2004.

19 Frank Plumpton Ramsey. On a problem in formal logic. *Proc. London Math. Soc.*, 30:264–286, 1930.

20 Dimitrios Vytiniotis, Thierry Coquand, and David Wahlstedt. Stop when you are almost-full – adventures in constructive termination. In *ITP*, pages 250–265, 2012.