

Formalising Formulas-as-Types-as-Objects

Qiao Haiyan

Department of Computing Science
Chalmers University of Technology and Göteborg University
412 96 Göteborg
qiao@cs.chalmers.se

Abstract. We describe a formalisation of the Curry-Howard-Lawvere correspondence between the natural deduction system for minimal logic, the typed lambda calculus and Cartesian closed categories. We formalise the type of natural deduction proof trees as a family of sets $\Gamma \vdash A$ indexed by the current assumption list Γ and the conclusion A and organise numerous useful lemmas about proof trees categorically. We prove categorical properties about proof trees up to (syntactic) identity as well as up to $\beta\eta$ -convertibility. We prove that our notion of proof trees is equivalent in an appropriate sense to more traditional representations of lambda terms.

The formalisation is carried out in the proof assistant ALF for Martin-Löf type theory.

1 Introduction

The background of the present paper is as follows. D. Čubrić, P. Dybjer and P. Scott discovered an elegant categorical decision method for equality in the free ccc [8]. This method was based on extracting an algorithm from some basic categorical facts related to the Yoneda lemma. To this end it was necessary to develop a certain version of constructive category theory, so called \mathcal{P} -category theory, which can be formalised inside a constructive metalanguage such as Martin-Löf type theory. It was also necessary to build various \mathcal{P} -categories from the syntax of the typed lambda calculus.

The paper by Čubrić, Dybjer, and Scott was written in an ordinary mathematical style and did not discuss formalisation in Martin-Löf type theory in detail. We have now succeeded in formalising a major part of Čubrić, Dybjer, and Scott [8] and thus is close to having verified the informal claims of that paper.

The present paper is an outgrowth of our formalisation effort. A key issue was how to formalise typed lambda terms inside Martin-Löf type theory and how to organise the necessary lemmas about them in a good way.

We shall here show how to formalise the formulas-as-types-as-objects correspondence (“Curry-Howard-Lawvere”) inside Martin-Löf type theory, a metalanguage which itself is based on the formulas-as-types correspondence. We consider the following basic part of the correspondence:

minimal logic	typed λ-calculus	cccs
formulas	types	objects
natural deduction proof trees	lambda terms	arrows
convertibility of proof trees	convertibility of lambda terms	equality of arrows

Let us first recall how these notions are typically presented informally in textbooks. To explain what a natural deduction proof tree is, one gives a number of rules for building such proof trees. An example is the \rightarrow -introduction rule

$$\frac{\begin{array}{c} [A] \\ \vdots \\ B \end{array}}{A \rightarrow B}$$

In general a proof tree may contain assumptions, such as $[A]$ above, and one has to explain what it means for an assumption to be discharged. The intention is to give enough informal explanations so that the reader will be able to build correct proof trees and to distinguish correct proof trees from incorrect ones.

It is clear that such an informal explanation can be modelled formally inside a standard mathematical metalanguage such as set theory, where relevant notions such as “tree” and “inductively defined collection” have well-known representations. But it is also clear that an informal presentation usually leaves many choices of such representations open. In traditional logic, not much attention has been paid to such choices. However, they are a central concern to someone who does formal metamathematics (perhaps on a machine). We shall not discuss them in detail in this paper. In this paper we shall use Martin-Löf type theory as a metalanguage.

Returning to natural deduction proof trees, we note that it has become increasingly common to replace the traditional notation for natural deduction proofs, as shown above, by a *sequent* notation, which makes the current list of undischarged hypotheses explicit. With this notation the rule of \rightarrow -introduction becomes

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \rightarrow B}$$

The introduction of this notation is helpful for mechanisation. We can view it either as a more explicit presentation of the “same” mathematical concept or a different, but equivalent, concept. The following obvious (?) point is important for formal metamathematics: if we, by *formal* proof, *formal* definition, etc., mean a proof *inside* a given metalanguage, then a textbook definition of a “formal” system is *informal*, except in the unusual case where a specific metalanguage is given and the formalisation of the system inside that metalanguage is given in

complete detail! So, even if the sequent-style presentation of minimal logic is as rigorous as anyone can require, it is still not formal. The process of formalising such an informal notion is, as all mathematical modelling, a subjective and experimental activity.

Consider now instead lambda terms, which according to the Curry-Howard correspondence can be used to denote natural deduction proofs. For example, the rule of \rightarrow -introduction in the typed lambda calculus becomes

$$\frac{\begin{array}{c} [x : A] \\ \vdots \\ b : B \end{array}}{\lambda x.b : A \rightarrow B}$$

where the term b , depending on x , represents the hypothetical proof of B from A , and the term $\lambda x.b$ represents the proof of $A \rightarrow B$ obtained by using \rightarrow -introduction on the hypothetical proof.

Again, as for the proof trees, when we want to formalise a textbook presentation of the typed lambda calculus, we have to make various choices. For example, should we first introduce a specific set of *raw terms* and then define *typing* as a relation between *raw terms*, *types and contexts*? Or should we try directly to inductively define a *family* of sets of terms, indexed by the types? We will show that these two ways are equivalent in the sense that we can build two isomorphic categories with families from such terms module $\beta\eta$ -conversion.

The rest of the paper is organised as follows. In section 2 we give the formal definitions (inside Martin-Löf type theory/ALF [16,14]) of minimal logic. In section 3 we introduce the notion of a simple category with families. In section 4 we show some properties of minimal logic structured as properties of three \mathcal{P} -categories with families: $(V, =)$ of proofs (up to syntactic equality) of the assumption rule; $(F, =)$ of proof trees (up to syntactic equality); and (F, \sim) of proof trees up to $\beta\eta$ -convertibility. In section 5 we formalise typed λ -calculus based on raw terms and prove that the three corresponding categories with families are isomorphic to those in section 4.

The ALF-files relating to the paper can be retrieved from the ftp-directory <ftp://ftp.cs.chalmers.se/pub/users/qiao/CHM/>.

2 Formalising Natural Deduction Proof Trees

2.1 Formulas and Contexts

Let X be a set of atomic formulas. The set Formula of formulas of minimal logic is then inductively generated by the following rules:

$$\frac{x : X}{\text{atom}(x) : \text{Formula}} \qquad \frac{A, B : \text{Formula}}{A \rightarrow B : \text{Formula}}$$

For simplicity we have assumed only one atom, i.e $X = \{O\}$. In ALF the definitions of the set Formula (Type is used instead of Formula) is given as the following sequence of typings

$\text{Type} \in \mathbf{Set}$
 $\text{O} \in \text{Type}$
 $\text{Arrow} \in (\text{Type}; \text{Type}) \text{Type}$

The set Context contains “snoc”-lists of formulas inductively generated by the following rules:

$$\begin{array}{c}
 [] : \text{Context} \\
 \hline
 \Gamma : \text{Context} \quad A : \text{Formula} \\
 \Gamma; A : \text{Context}
 \end{array}$$

Context is given as follows in ALF:

$\text{Context} \in \mathbf{Set}$
 $\text{Nil} \in \text{Context}$
 $\text{Cons} \in (\Gamma \in \text{Context}; A \in \text{Type}) \text{Context}$

Here $(A)B$ is ALF’s notation for the type of functions from A to B . $(A_0; A_1)B$ is an abbreviation of $(A_0)(A_1)B$, the type of binary curried functions from A_0 and A_1 to B . Furthermore, ALF uses $(x : A)B[x]$ for the *dependent function type*, that is, the type of functions which map an object $x : A$ to an object of $B[x]$, where $B[x]$ is a type which may depend on x . The notation $(x_0 : A_0; x_1 : A_1[x_0])B[x_0, x_1]$ is an abbreviation of $(x_0 : A_0)(x_1 : A_1[x_0])B[x_0, x_1]$, the type of binary curried functions which map an $x_0 : A_0$ and an $x_1 : A_1[x_0]$ to an object of $B[x_0, x_1]$.

2.2 The Assumption Rule

In the sequent-style presentation of minimal logic, the assumption rule lets us conclude that

$$\Gamma \vdash A$$

provided A is one of the formulas in Γ .

Below, we shall formally define the family of sets $\text{Term}(\Gamma, A)$ of natural deduction proof trees with conclusion A and assumption list Γ . One of its introduction rules will be the assumption rule in the following form:

$$\frac{v : \text{Var}(\Gamma, A)}{\text{var}(v) : \text{Term}(\Gamma, A)}$$

where $\text{Var}(\Gamma, A)$ is the set of proofs that A is a member of the list Γ . To improve readability we borrow the traditional infix notation and introduce the definitions

$$\begin{aligned}
 A \in \Gamma &= \text{Var}(\Gamma, A) \\
 \Gamma \vdash A &= \text{Term}(\Gamma, A)
 \end{aligned}$$

so that the assumption rule can be rewritten as

$$\frac{v : A \in \Gamma}{\text{var}(v) : \Gamma \vdash A}$$

Read: “if v is a proof that A is a member of Γ , then $\text{var}(v)$ is a proof that A follows from Γ by assumption.”

The membership relation is inductively generated by the following two rules

$$\frac{}{0_{\Gamma, A} : A \in (\Gamma; A)} \qquad \frac{B : \text{Formula} \quad v : A \in \Gamma}{s_{\Gamma, A, B}(v) : A \in (\Gamma; B)}$$

In ALF, the family $A \in \Gamma$ is formalised as follows:

$\text{Var} \in (\Gamma \in \text{Context}; A \in \text{Type}) \mathbf{Set}$
 $\text{Zero} \in (\Gamma \in \text{Context}; A \in \text{Type}) \text{Var}(\text{Cons}(\Gamma, A), A)$
 $\text{Succ} \in (\Gamma \in \text{Context}; A, B \in \text{Type}; \text{Var}(\Gamma, A)) \text{Var}(\text{Cons}(\Gamma, B), A)$

To make it more readable, like in the informal notations, we will suppress some uninteresting arguments in the constructors Zero and Succ:

$\text{Var} \in (\Gamma \in \text{Context}; A \in \text{Type}) \mathbf{Set}$
 $\text{Zero} \in \text{Var}(\text{Cons}(\Gamma, A), A)$
 $\text{Succ} \in (\text{Var}(\Gamma, A)) \text{Var}(\text{Cons}(\Gamma, B), A)$

2.3 Proof Trees

The following rules inductively generate the indexed family of sets of proof trees:

$$\frac{v : A \in \Gamma}{\text{var}(v) : \Gamma \vdash A} \qquad \frac{b : \Gamma; A \vdash B}{\lambda(b) : \Gamma \vdash A \rightarrow B} \qquad \frac{c : \Gamma \vdash A \rightarrow B \quad a : \Gamma \vdash A}{\text{app}(c, a) : \Gamma \vdash B}$$

The family $\Gamma \vdash A$ is implemented in ALF in the following way:

$\text{Term} \in (\Gamma \in \text{Context}; A \in \text{Type}) \mathbf{Set}$
 $\text{Variable} \in (v \in \text{Var}(\Gamma, A)) \text{Term}(\Gamma, A)$
 $\text{Lambda} \in (b \in \text{Term}(\text{Cons}(\Gamma, A), B)) \text{Term}(\Gamma, \text{Arrow}(A, B))$
 $\text{Application} \in (u \in \text{Term}(\Gamma, \text{Arrow}(A, B)); v \in \text{Term}(\Gamma, A)) \text{Term}(\Gamma, B)$

Again, we have suppressed some arguments of the constructors and the corresponding premises of the inference rule. We will often do the same in the sequel without mentioning it.

3 Simple Categories with Families

We shall prove various categorical properties of our formal natural deduction proof trees. There are several reasons for this. Firstly, it turns out that many of the operations and lemmas needed for developing the meta-theory of natural deduction have natural categorical interpretations. In other words, category theory can be used as a structuring device for a library of such lemmas. Secondly, a comparison with categorical notions will highlight the “pseudo-categorical” nature of our formalisation. Thirdly, we want to show that there is more to say about the correspondence between categories, typed lambda calculus and natural deduction proof trees, than the well-known correspondence between the typed lambda calculus and Cartesian closed categories.

The point is that some of these properties will hold for proof trees up to syntactic identity, others will hold up to $\beta\eta$ -convertibility. It turns out that

these properties can be organised in an elegant categorical way, using the notion of *category with families (cwf)* [10].

Cwfs were introduced by Dybjer [10] to be an appropriate categorical notion of model for type-dependency. Similar notions have also been considered: Cartmell's *contextual categories* [5], Seely's *locally Cartesian closed categories* [20], and Pitts' *categories with fibration* [18]. Cwfs are closer to the traditional syntax of dependent types, and at the same time they are completely algebraic and have a nice categorical description.

A cwf has a family $\text{Ty}(\Gamma)$ of types in context Γ as part of its structure. Since we here do not consider dependent types, it suffices to consider cwfs, where $\text{Ty}(\Gamma)$ does not depend on Γ . We call such cwfs “simple”. Similar notions are Obtulowicz's *Church algebraic theory* [17], Jacobs' *indexed categories* [13], Altenkirch, Hofmann and Streicher's *contextual CCCs* [2], and etc.

Definition 1 *A simple cwf consists of the following parts:*

- *A base category \mathcal{C} . Its objects are called contexts and its morphisms are called substitutions. We write $\Delta \rightarrow \Gamma$ for the hom-set of morphisms (substitutions) with source Δ and target Γ .*
- *A set Ty of types.*
- *A presheaf $\text{Tm}^A : \mathcal{C}^{op} \rightarrow \text{Set}$ for each type A . The object part of the functor associates to a context Γ the set $\text{Tm}^A(\Gamma)$ (or $\text{Tm}(\Gamma, A)$) of terms and the arrow part of the functor associates to a substitution γ the operation which applies γ to a term a to get the result of the substitution $a[\gamma]$.*
- *A terminal object \square of \mathcal{C} called the empty context.*
- *A context-extension operation which to a context Γ and a type A associates a context $\Gamma; A$. Furthermore there is a substitution $p : \Gamma; A \rightarrow \Gamma$ (the first projection) and a term q in the set $\text{Tm}(\Gamma; A, A)$ (the second projection). The following universal property holds: for each context Δ , substitution $\gamma : \Delta \rightarrow \Gamma$, and term $a : \text{Tm}(\Delta, A)$, there is a unique substitution $\theta = \langle \gamma, a \rangle : \Delta \rightarrow \Gamma; A$, such that $p \circ \theta = \gamma$ and $q[\theta] = a$.*

Simple cwfs can be axiomatised by taking the axiomatisation of general cwfs from Dybjer [10] and removing type-dependency. More information on cwfs can be found in Hofmann [11].

We will refer to a cwf as a structure $(\mathcal{C}, \text{Ty}, \text{Tm})$.

\mathcal{P} -cwfs. When we formalise category theory in Martin-Löf type theory we shall follow Čubrić, Dybjer, and Scott [8] and use \mathcal{P} -categories. A \mathcal{P} -category is a category where we replace the equality on arrows by a partial equivalence relation (per) systematically. Other notions in category theory will be prefixed by \mathcal{P} -when they are used in the context of \mathcal{P} -categories.

A cwf $(\mathcal{C}, \text{Ty}, \text{Tm})$ is called a \mathcal{P} -cwf if the base category \mathcal{C} is \mathcal{P} -category and the set $\text{Tm}(\Gamma, A)$ is a \mathcal{P} -set, i.e. a set with a partial equivalence relation.

We will refer to a \mathcal{P} -cwf as a structure $(\mathcal{C}, \sim, \text{Ty}, \text{Tm})$ or simply as a “pair” (\mathcal{C}, \sim) of the base category when we want to emphasise the per-structure on arrows.

Definition 2 Let $(C, \text{Ty}_C, \text{Tm}_C)$ and $(D, \text{Ty}_D, \text{Tm}_D)$ be the two \mathcal{P} -simple cwf's. A morphism from $(C, \text{Ty}_C, \text{Tm}_C)$ to $(D, \text{Ty}_D, \text{Tm}_D)$ is a triple (F, T, θ) :

$F : C \rightarrow D$ is a \mathcal{P} -functor;

$T : \text{Ty}_C \rightarrow \text{Ty}_D$ is a function and

$\theta : \text{Tm}_C^A \rightarrow \text{Tm}_D^A$ is a \mathcal{P} -natural transformation, i.e.

$\theta(a[f]) \sim \theta(a')[F(f')]$ if $a \sim a'$ and $f \sim f'$, where the notation \sim is overloaded.

Furthermore, the terminal object and context comprehension are preserved up to isomorphism.

Definition 3 Let $(C, \text{Ty}_C, \text{Tm}_C)$ and $(D, \text{Ty}_D, \text{Tm}_D)$ be two \mathcal{P} -cwf's,

$(F, T, \theta) : (C, \text{Ty}_C, \text{Tm}_C) \rightarrow (D, \text{Ty}_D, \text{Tm}_D)$ and

$(F', T', \theta') : (D, \text{Ty}_D, \text{Tm}_D) \rightarrow (C, \text{Ty}_C, \text{Tm}_C)$ be a pair of morphisms.

These morphisms form an isomorphism of cwf's if and only if

1. F and F' form an isomorphism between the \mathcal{P} -categories C and D .
2. for any $A \in |\text{Ty}_C|$, $T'(T(A)) \sim A$ and for any $A \in |\text{Ty}_D|$, $T(T'(A)) \sim A$
3. for any $a \in |\text{Tm}_C^A(\Gamma)|$, $\theta'(\theta(a)) \sim a$ and
for any $a \in |\text{Tm}_D^A(\Gamma)|$, $\theta(\theta'(a)) \sim a$

where \sim refers to the per on the corresponding \mathcal{P} -set.

It is generally agreed that it is too restrictive to stipulate that the category laws hold up to the built-in identity of Martin-Löf type theory. This is because there is no quotienting operation. Most authors who have developed category theory inside Martin-Löf type theory have therefore used what could be called \mathcal{E} -categories, where one requires a *total* equivalence relation on hom-sets. However, the extra generality of \mathcal{P} -categories was crucial for the categorical normalisation proof of Čubrić, Dybjer, and Scott[8].

To implement a simple \mathcal{P} -cwf in ALF one defines a sequence of typings:

$\text{Pcwf} \equiv [\text{Cont} \in \mathbf{Set};$

$\text{ArrCon} \in (\text{Cont}; \text{Cont}) \mathbf{Set};$

$\text{EqArr} \in (\Gamma, \Delta \in \text{Cont}; \text{ArrCon}(\Gamma, \Delta); \text{ArrCon}(\Gamma, \Delta)) \mathbf{Set};$

$\text{Cid} \in (\Gamma \in \text{Cont}) \text{ArrCon}(\Gamma, \Gamma);$

$\text{Ccom} \in (\Gamma, \Delta, F \in \text{Cont}; \text{ArrCon}(\Delta, F); \text{ArrCon}(\Gamma, \Delta)) \text{ArrCon}(\Gamma, F);$

...

$\text{Typ} \in \mathbf{Set};$

$\text{Tm} \in (\text{Cont}; \text{Typ}) \mathbf{Set};$

$\text{Sub} \in (\Gamma, \Delta \in \text{Cont}; A \in \text{Typ}; \text{Tm}(\Delta, A); \text{ArrCon}(\Gamma, \Delta)) \text{Tm}(\Gamma, A);$

...]

An ALF-context can be used in two ways: one can both instantiate such a context to obtain a particular \mathcal{P} -cwf, and assume it and reason from the assumption that one has an arbitrary \mathcal{P} -cwf (as in the proof of freeness).

4 Categorical Properties of Proof Trees

4.1 The Assumption Rule

We shall now define the \mathcal{P} -cwf $(V, =)$, the terms of which are proofs of the assumption rule up to the syntactic equality.

We define the set $\Delta \supseteq \Gamma$ of sequences of proofs of the assumption rule. The reason for the notation is that the elements $\Delta \supseteq \Gamma$ may be thought of as proofs that the formulas in Γ is a subset of those of Δ . These proofs will also serve as arrows of the base category of $(V, =)$. The introduction rules are

$$\frac{\Delta : \text{Context}}{\langle \rangle : \Delta \supseteq []} \qquad \frac{vs : \Delta \supseteq \Gamma \quad v : A \in \Delta}{\langle vs, v \rangle : \Delta \supseteq \Gamma; A}$$

In ALF $\Delta \supseteq \Gamma$ is represented by $\text{VarList}(\Delta, \Gamma)$:

$\text{VarList} \in (\Delta, \Gamma \in \text{Context}) \mathbf{Set}$

$\text{Nilvar} \in (\Delta \in \text{Context}) \text{VarList}(\Delta, \text{Nil})$

$\text{ConsVar} \in (vars \in \text{VarList}(\Delta, \Gamma); v \in \text{Var}(\Delta, A)) \text{VarList}(\Delta, \text{Cons}(\Gamma, A))$

Two arrows vs_1, vs_2 are equal, denoted as $vs_1 = vs_2$, if they are identical point-wise:

$\text{EqualVars} \in (vs_1, vs_2 \in \text{VarList}(\Delta, \Gamma)) \mathbf{Set}$

$\text{IntroNilVars} \in \text{EqualVars}(\text{Nilvar}(\Delta), \text{Nilvar}(\Delta))$

$\text{IntroConVars} \in (p_1 \in \text{EqualVars}(vs_1, vs_2);$

$p_2 \in \text{I}(v_1, v_2)$

$) \text{EqualVars}(\text{ConsVar}(vs_1, v_1), \text{ConsVar}(vs_2, v_2))$

where the identity type I is introduced as follows:

$\text{I} \in (a, b \in A) \mathbf{Set}$

$r \in (a \in A) \text{I}(a, a)$

Moreover, we define a few operations which are parts of the structure of a \mathcal{P} -cwf.

First we have the operation which maps $v : A \in \Gamma$ and $vs : \Delta \supseteq \Gamma$ to $v[vs] : A \in \Delta$:

$\text{ProjectVars} \in (vs \in \text{VarList}(\Delta, \Gamma); v \in \text{Var}(\Gamma, A)) \text{Var}(\Delta, A)$

$\text{ProjectVars}(\text{ConsVar}(vars, v_1), \text{Zero}) \equiv v_1$

$\text{ProjectVars}(\text{ConsVar}(vars, v_1), \text{Succ}(h)) \equiv \text{ProjectVars}(vars, h)$

Then we have the composition operation:

$\text{CompositVars} \in (vs \in \text{VarList}(\Phi, \Delta); ws \in \text{VarList}(\Delta, \Gamma)) \text{VarList}(\Phi, \Gamma)$

$\text{CompositVars}(vs, \text{Nilvar}(-)) \equiv \text{Nilvar}(\Phi)$

$\text{CompositVars}(vs, \text{ConsVar}(vars, v)) \equiv \text{ConsVar}(\text{CompositVars}(vs, vars), \text{ProjectVars}(vs, v))$

where the symbol $_$ is an argument which ALF can infer.

Finally, the identity $\text{id}_\Gamma : \Gamma \supseteq \Gamma$ and the first projection $p_{\Gamma, A} : \Gamma; A \supseteq \Gamma$:

$\text{IdVars} \in (\Gamma \in \text{Context}) \text{VarList}(\Gamma, \Gamma)$

$\text{IdVars}(\text{Nil}) \equiv \text{Nilvar}(\text{Nil})$

$\text{IdVars}(\text{Cons}(\Gamma, A)) \equiv \text{ConsVar}(\text{LiftVars}(A, \text{IdVars}(\Gamma)), \text{Zero})$

ProjecVars $\in (\Gamma \in \text{Context}; A \in \text{Type}) \text{VarList}(\text{Cons}(\Gamma, A), \Gamma)$
 ProjecVars(Γ, A) \equiv LiftVars($A, \text{IdVars}(\Gamma)$)

where LiftVars(A, vs) extends the assumption list Γ of vs by adding a formula A on Γ :

LiftVars $\in (A \in \text{Type}; \text{vs} \in \text{VarList}(\Gamma, \Delta)) \text{VarList}(\text{Cons}(\Gamma, A), \Delta)$
 LiftVars($A, \text{Nilvar}(-)$) \equiv Nilvar($\text{Cons}(\Gamma, A)$)
 LiftVars($A, \text{ConsVar}(\text{vars}, v)$) \equiv ConsVar(LiftVars(A, vars), Succ(v))

Theorem 1 ($V, =, \text{Type}, \text{Var}$) is a simple \mathcal{P} -cwf, which we will refer to $(V, =)$. Furthermore, for any \mathcal{P} -cwf \mathcal{D} and any function $J : \text{Type} \rightarrow \text{Ty}_{\mathcal{D}}$ there is a unique (up to \mathcal{P} -natural isomorphisms) \mathcal{P} -cwf-morphism $\llbracket - \rrbracket : (V, =) \rightarrow \mathcal{D}$.

4.2 Proof Trees up to Syntactic Identity

We shall now define the \mathcal{P} -cwf $(F, =)$, the terms of which are proof trees of minimal logic up to syntactic identity.

We first define the sets $\Delta \rightarrow \Gamma$ of sequences of proof trees, which will be the substitutions of $(F, =)$. The introduction rules are

$$\frac{\Delta : \text{Context}}{\langle \rangle : \Delta \rightarrow []} \qquad \frac{as : \Delta \rightarrow \Gamma \quad a : \Delta \vdash A}{\langle as, a \rangle : \Delta \rightarrow \Gamma; A}$$

In ALF TermList(Δ, Γ) implements $\Delta \rightarrow \Gamma$:

TermList $\in (\Delta, \Gamma \in \text{Context}) \mathbf{Set}$
 NilList \in TermList(Δ, Nil)
 ConsList $\in (as \in \text{TermList}(\Delta, \Gamma); a \in \text{Term}(\Delta, A)) \text{TermList}(\Delta, \text{Cons}(\Gamma, A))$

The equality '=' is defined by the identity type component-wise:

IonFx $\in (ts_1, ts_2 \in \text{TermList}(\Delta, \Gamma)) \mathbf{Set}$
 IntroNilIonFx \in IonFx(NilList, NilList)
 IntroConIonFx $\in (p_1 \in \text{IonFx}(ts_1, ts_2);$
 $p_2 \in \mathbf{I}(t_1, t_2)$
 $) \text{IonFx}(\text{ConsList}(ts_1, t_1), \text{ConsList}(ts_2, t_2))$

Moreover, we define a few operations which are part of the structure of a \mathcal{P} -cwf.

The substitution operation $-[-]$ is defined by the following constant:

Subst $\in (ts \in \text{TermList}(\Gamma, \Delta); b \in \text{Term}(\Delta, B)) \text{Term}(\Gamma, B)$
 Subst(ConsList(bs, b), Variable(Zero)) $\equiv b$
 Subst(ConsList(bs, b), Variable(Succ(h))) \equiv Subst(bs , Variable(h))
 Subst(ts , Lambda(b_I)) \equiv Lambda(Subst(Lift(ts), b_I))
 Subst(ts , Application(u, v)) \equiv Application(Subst(ts, u), Subst(ts, v))

where Lift is defined below in such a way that the bound variable Variable(Zero) in b remains unchanged and other free variables in ts are lifted because they are pushed inside one λ :

lift $\in (n \in \mathbf{N}; \text{lterm}(n)) \text{lterm}(\text{succ}(n))$
 lift $\equiv [n, h]\text{rename}(n, h, \text{succ}(n), \text{projec}(n))$

where $\text{ExtendCon}(A, \text{bs})$ applies the weakening rule $\text{LiftTerm}(A, -)$ to bs point-wise:

$$\begin{aligned} \text{LiftTerm} &\in (A \in \text{Type}; b \in \text{Term}(\Gamma, B)) \text{Term}(\text{Cons}(\Gamma, A), B) \\ \text{LiftTerm}(A, b) &\equiv \text{SubVars}(\text{ProjecVars}(\Gamma, A), b) \end{aligned}$$

and the operation $\text{SubVars}(vs, b)$ (the Thinning Lemma) can be read as: if we have a proof $vs : \Delta \supseteq \Gamma$ and a proof tree $b : \Gamma \vdash B$, then we can construct a proof tree $\text{SubVars}(vs, b) : \Delta \vdash B$:

$$\begin{aligned} \text{SubVars} &\in (vs \in \text{VarList}(\Delta, \Gamma); b \in \text{Term}(\Gamma, B)) \text{Term}(\Delta, B) \\ \text{SubVars}(\text{ConsVar}(vars, v), \text{Variable}(\text{Zero})) &\equiv \text{Variable}(v) \\ \text{SubVars}(\text{ConsVar}(vars, v), \text{Variable}(\text{Succ}(h))) &\equiv \text{SubVars}(vars, \text{Variable}(h)) \\ \text{SubVars}(vs, \text{Lambda}(b_l)) &\equiv \\ &\quad \text{Lambda}(\text{SubVars}(\text{ConsVar}(\text{LiftVars}(A, vs), \text{Zero}), b_l)) \\ \text{SubVars}(vs, \text{Application}(u, v)) &\equiv \text{Application}(\text{SubVars}(vs, u), \text{SubVars}(vs, v)) \end{aligned}$$

Then the composition is defined as

$$\begin{aligned} \text{CompositTms} &\in (s \in \text{TermList}(\Phi, \Delta); t \in \text{TermList}(\Delta, \Gamma)) \text{TermList}(\Phi, \Gamma) \\ \text{CompositTms}(s, \text{NilList}) &\equiv \text{NilList} \\ \text{CompositTms}(s, \text{ConsList}(bs, b)) &\equiv \text{ConsList}(\text{CompositTms}(s, bs), \text{Subst}(s, b)) \end{aligned}$$

The identity and the first projection are defined as:

$$\begin{aligned} \text{Fx_Id} &\in (\Gamma \in \text{Context}) \text{TermList}(\Gamma, \Gamma) \\ \text{Fx_Id}(\Gamma) &\equiv \text{mapVars}(\Gamma, \Gamma, \text{IdVars}(\Gamma)) \end{aligned}$$

$$\begin{aligned} \text{Fx_p} &\in (\Gamma \in \text{Context}; A \in \text{Type}) \text{TermList}(\text{Cons}(\Gamma, A), \Gamma) \\ \text{Fx_p}(\Gamma, A) &\equiv \text{mapVars}(\text{Cons}(\Gamma, A), \Gamma, \text{ProjecVars}(\Gamma, A)) \end{aligned}$$

where

$$\begin{aligned} \text{mapVars} &\in (\Gamma, \Delta \in \text{Context}; vs \in \text{VarList}(\Gamma, \Delta)) \text{TermList}(\Gamma, \Delta) \\ \text{mapVars}(\Gamma, -, \text{Nilvar}(-)) &\equiv \text{NilList} \\ \text{mapVars}(\Gamma, -, \text{ConsVar}(vars, v)) &\equiv \text{ConsList}(\text{mapVars}(\Gamma, \Delta, vars), \text{Variable}(v)) \end{aligned}$$

Theorem 2 $(F, =, \text{Type}, \text{Term})$ is a \mathcal{P} -cwf.

Čubrić, Dybjer, and Scott also utilized that the base \mathcal{P} -category of $(F, =)$ had some further categorical properties, which we have formalised.

Theorem 3 *The base category of $(F, =, \text{Type}, \text{Term})$ has finite \mathcal{P} -products $(\Gamma \times \Delta)$ and “pre-exponentials”, i.e. given objects Γ, Δ we have an object Δ^Γ ; an evaluation arrow $\epsilon : \Delta^\Gamma \times \Gamma \rightarrow \Delta$; and a currying operation $\gamma^* : \Gamma \rightarrow \Theta^\Delta$ for $\gamma : \Gamma \times \Delta \rightarrow \Theta$ such that $\gamma^* \circ \delta = (\gamma \circ \langle \delta \circ \text{fst}, \text{snd} \rangle)^*$. (Note that “=” denotes syntactic identity here, so that we cannot expect to have proper exponentials.)*

4.3 Categorical Properties of Proof Trees up to $\beta\eta$ -Equality

We shall now consider proof trees up to $\beta\eta$ -convertibility. This is a family of binary relations $\sim_{\Gamma, A}$ (indexed by contexts Γ and formulas A) between proof trees in $\Gamma \vdash A$. It is inductively generated by the following rules (where we have dropped the indices of \sim):

Per-rules:

$$\frac{a \sim a'}{a' \sim a} \qquad \frac{a \sim a' \quad a' \sim a''}{a \sim a''}$$

Congruence:

$$\text{var}(v) \sim \text{var}(v) \qquad \frac{c \sim c' \quad a \sim a'}{\text{app}(c, a) \sim \text{app}(c', a')} \qquad \frac{b \sim b'}{\lambda(b) \sim \lambda(b')}$$

 β -rule:

$$\frac{b : \Gamma; A \vdash B \quad a : \Gamma \vdash A}{\text{app}(\lambda(b), a) \sim b[\langle \text{Fx_Id}, a \rangle]}$$

 η -rule:

$$\frac{c : \Gamma \vdash A \rightarrow B}{c \sim \lambda(\text{app}(\text{LiftTerm}(c), \text{var}(0)))}$$

The $\beta\eta$ equality is implemented as:

`EqualTerm` $\in (t_1, t_2 \in \text{Term}(\Gamma, A))$ **Set**

Theorem 4 *Let $(F, \sim, \text{Type}, \text{Term})$ be defined in the same way as $(F, =, \text{Type}, \text{Term})$ except that the per on terms is $\beta\eta$ -convertibility and the per on substitutions is $\beta\eta$ -convertibility extended point-wise. Then $(F, \sim, \text{Type}, \text{Term})$ is a \mathcal{P} -cwf.*

We have also formalised the proof of the following theorem, which was used by Čubrić, Dybjer, and Scott [8].

Theorem 5 *The base category (F, \sim) is a free \mathcal{P} -ccc.*

5 The Correspondence between Different Formalisations

In this section we shall show the equivalence between our representation and a representation using raw terms à la de Bruijn.

5.1 Bounded de Bruijn Indices

We will define raw terms using bounded de Bruijn-indices which make the number of free variables explicit [9].

Our bounded de Bruijn-indices are elements of finite sets N'_n indexed by $n \in \mathbb{N}$ with the following introduction rules:

$$\frac{n : \mathbb{N}}{0' : N'_{s(n)}} \qquad \frac{n : \mathbb{N} \quad i : N'_n}{s'(i) : N'_{s(n)}}$$

The family of bounded de Bruijn-indices is formalised directly in ALF:

$N' \in (\mathbf{N}) \mathbf{Set}$

$0' \in (n \in \mathbf{N}) N'(\text{succ}(n))$

$s' \in (n \in \mathbf{N}; N'(n)) N'(\text{succ}(n))$

where the set \mathbf{N} is formalised as:

$\mathbf{N} \in \mathbf{Set}$

$\text{zero} \in \mathbf{N}$

$\text{succ} \in (n \in \mathbf{N}) \mathbf{N}$

In this section we will define a \mathcal{P} -cwf $(N'_{\text{typa}}, =)$ of bounded de Bruijn indices, the analogue of the $(V, =)$, and prove that the cwf $(N'_{\text{typa}}, =)$ is isomorphic to $(V, =)$. We introduce typing rules for the bounded de Bruijn-indices $i :: A \in \Gamma$, meaning that the index $i \in N'_{|\Gamma|}$ ($|\Gamma|$ is the number of formulae in the context Γ) has type A in the context Γ :

$$\frac{}{0' :: A \in \Gamma; A} \qquad \frac{i :: A \in \Gamma}{s'(i) :: A \in \Gamma; B}$$

These rules define a relation between contexts, types and de Bruijn-indices. The relation $i :: A \in \Gamma$ can be formalised by $N'_{\text{typa}}(\Gamma, A, i)$ in the following way:

$N'_{\text{typa}} \in (\Gamma \in \text{Context}; A \in \text{Type}; N'(\text{Cont-}\mathbf{N}(\Gamma))) \mathbf{Set}$

$N'_{\text{typa}}(\text{Cons}(\Delta, A_l), A, 0'(-)) \equiv \mathbf{I}(A_l, A)$

$N'_{\text{typa}}(\text{Cons}(\Delta, A_l), A, s'(-, h_l)) \equiv N'_{\text{typa}}(\Delta, A, h_l)$

where $\text{Cont-}\mathbf{N}(\Gamma)$ is the length of Γ .

We will say that the index i is typable if $i :: A \in \Gamma$.

Now we can define the set $(\Sigma i : N'_{|\Gamma|})(i :: A \in \Gamma)$ in ALF as follows:

$\text{Term-Ns} \in (\text{Context}; \text{Type}) \mathbf{Set}$

$\text{term-Ns} \in (\Gamma \in \text{Context}; A \in \text{Type}; i \in N'(\text{Cont-}\mathbf{N}(\Gamma)); N'_{\text{typa}}(\Gamma, A, i)) \text{Term-Ns}(\Gamma, A)$

where an element of the set $\text{Term-Ns}(\Gamma, A)$ is a pair of an index $i \in N'_{|\Gamma|}$ and a proof that the index i has type A under the context Γ , i.e. $i :: A \in \Gamma$.

A tuple of indices $js = (i_1, \dots, i_n) : N'^n_m$ is typable from the context Γ to the context Δ , denoted as $js :: \Gamma \supseteq \Delta$, if $\Delta = A_1, \dots, A_n$ and $i_k :: A_k \in \Gamma$ for $k = 1, \dots, n$:

$$\frac{}{\langle \rangle :: \Gamma \supseteq []} \qquad \frac{js :: \Gamma \supseteq \Delta \quad i :: A \in \Gamma}{\langle js, i \rangle :: \Gamma \supseteq \Delta; A}$$

Then we define the family $(\Sigma js : N'_{|\Delta|})(js :: \Gamma \supseteq \Delta)$ of typable lists of indices in ALF:

$\text{Typa-Ns} \in (\Gamma, \Delta \in \text{Context}) \mathbf{Set}$

$\text{typa-Ns} \in (js \in \text{Tuple}(N'(\text{Cont-}\mathbf{N}(\Gamma)), \text{Cont-}\mathbf{N}(\Delta)); \text{Typa-Ns-2}(\Gamma, \Delta, js)) \text{Typa-Ns}(\Gamma, \Delta)$

where $\text{Typa-Ns-2}(\Gamma, \Delta, js)$ implements the relation $js :: \Gamma \supseteq \Delta$.

Equality “=” on the set $\text{Typa-Ns}(\Gamma, \Delta)$ is syntactic equality on the first part of the set:

$\text{per-Ns} \in (\Gamma, \Delta \in \text{Context}; \text{Typa-Ns}(\Gamma, \Delta); \text{Typa-Ns}(\Gamma, \Delta)) \mathbf{Set}$

$\text{per-Ns}(\Gamma, \Delta, \text{typa-Ns}(js, h_2), \text{typa-Ns}(js_l, h)) \equiv \mathbf{I}(js, js_l)$

Theorem 6 $(N'_{\text{typa}}, =, \text{Type}, \text{Term_Ns})$ is a \mathcal{P} -cwf. Furthermore, this \mathcal{P} -cwf is isomorphic to the \mathcal{P} -cwf $(V, =, \text{Type}, \text{Var})$.

We here outline how the isomorphism is proved in ALF. This is proved by establishing a pair of morphism between $(V, =)$ and $(N'_{\text{typa}}, =)$ and by checking they form an isomorphism.

First we define the morphism $(H, T, \text{morph_Var_N}')$ from $(V, =)$ to $(N'_{\text{typa}}, =)$, where

- T is the identity function from Type to Type ;
- the natural transformation $\text{morph_Var_N}'$ is defined as follows:

$\text{morph_Var_N}' \in (\Gamma \in \text{Context}; A \in \text{Type}; \text{Var}(\Gamma, A)) \text{Term_Ns}(\Gamma, A)$
 $\text{morph_Var_N}'(\Gamma, A, h) \equiv \text{term_Ns}(\Gamma, A, \text{Var_N}'(\Gamma, A, h), \text{Var_N}'\text{-typa}(\Gamma, A, h))$

where $\text{Var_N}'(\Gamma, A, \gamma)$ returns an untyped index for a typed index $\gamma : A \in \Gamma$:

$\text{Var_N}' \in (\Gamma \in \text{Context}; A \in \text{Type}; v \in \text{Var}(\Gamma, A)) N'(\text{Cont_N}(\Gamma))$
 $\text{Var_N}'(-, A, \text{Zero}) \equiv 0'(\text{Cont_N}(\Gamma))$
 $\text{Var_N}'(-, A, \text{Succ}(h)) \equiv s'(\text{Cont_N}(\Gamma), \text{Var_N}'(\Gamma, A, h))$

and $\text{Var_N}'\text{-typa}(\Gamma, A, h)$ is a proof that this untyped index is typable, i.e $\text{Var_N}'(\Gamma, A, h) :: A \in \Gamma$.

- the functor H consists of the identity map on Context (the set of objects) and the following map on arrows:

$\text{morph_Vx_Ns} \in (\Gamma, \Delta \in \text{Context}; \text{VarList}(\Gamma, \Delta)) \text{Typa_Ns}(\Gamma, \Delta)$
 $\text{morph_Vx_Ns}(\Gamma, \Delta, h) \equiv$
 $\text{typa_Ns}(\text{VarList_Tup}(\Gamma, \Delta, h), \text{VarList_Tup_typa}(\Gamma, \Delta, h))$

where VarList_Tup applies $\text{Var_N}'$ component-wise to $vs \in \Gamma \supseteq \Delta$ and returns a tuple:

$\text{VarList_Tup} \in (\Gamma, \Delta \in \text{Context};$
 $vs \in \text{VarList}(\Gamma, \Delta)$
 $) \text{Tuple}(N'(\text{Cont_N}(\Gamma)), \text{Cont_N}(\Delta))$
 $\text{VarList_Tup}(\Gamma, -, \text{Nilvar}(-)) \equiv \text{one}$
 $\text{VarList_Tup}(\Gamma, -, \text{ConsVar}(vars, v)) \equiv$
 $\text{pair}(\text{VarList_Tup}(\Gamma, \Delta, vars), \text{Var_N}'(\Gamma, A, v))$

and $\text{Varlist_Tup_typa}(\Gamma, \Delta, vs)$ is a proof object that the returned tuple is typable: $\text{VarList_Tup}(\Gamma, \Delta, vs) :: \Gamma \supseteq \Delta$.

In the other direction, we define a morphism $(H', T, \text{morph_N}'\text{-Var})$ from $(N'_{\text{typa}}, =)$ to $(V, =)$, where

- the natural transformation $\text{morph_N}'\text{-Var}$ is defined as:

$\text{morph_N}'\text{-Var} \in (\Gamma \in \text{Context}; A \in \text{Type}; \text{Term_Ns}(\Gamma, A)) \text{Var}(\Gamma, A)$
 $\text{morph_N}'\text{-Var}(\Gamma, A, \text{term_Ns}(-, -, i, h_i)) \equiv N'\text{-Var}(\Gamma, A, i, h_i)$

$N'_{\text{-Var}} \in (\Gamma \in \text{Context};$
 $A \in \text{Type};$
 $p \in N'(\text{Cont-N}(\Gamma));$
 $N'_{\text{-typa}}(\Gamma, A, p)$
 $) \text{Var}(\Gamma, A)$
 $N'_{\text{-Var}}(\text{Cons}(\Gamma, -), A, 0'(-), r(-)) \equiv \text{Zero}$
 $N'_{\text{-Var}}(\text{Cons}(\Gamma, A_i), A, s'(-, h_i), h) \equiv \text{Succ}(N'_{\text{-Var}}(\Gamma, A, h_i, h))$
 – the functor H' consists of the identity map on Context , and the following map on arrows, which is defined by extending $N'_{\text{-Var}}$ component-wise:
 $\text{morph-Ns-Vx} \in (\Gamma, \Delta \in \text{Context}; \text{Typa-Ns}(\Gamma, \Delta)) \text{VarList}(\Gamma, \Delta)$
 $\text{morph-Ns-Vx}(\Gamma, \Delta, h) \equiv \text{Ns-Tup-Vars}(\Gamma, \Delta, h)$
 $\text{Ns-Tup-Vars} \in (\Gamma, \Delta \in \text{Context}; \text{Typa-Ns}(\Gamma, \Delta)) \text{VarList}(\Gamma, \Delta)$
 $\text{Ns-Tup-Vars}(\Gamma, \text{Nil}, \text{typa-Ns}(as, h_i)) \equiv \text{Nilvar}(\Gamma)$
 $\text{Ns-Tup-Vars}(\Gamma, \text{Cons}(\Gamma, A), \text{typa-Ns}(\text{pair}(a, b), \text{andIntr}(h, h_2))) \equiv$
 $\text{ConsVar}(\text{Ns-Tup-Vars}(\Gamma, \Gamma, \text{typa-Ns}(a, h)), N'_{\text{-Var}}(\Gamma, A, b, h_2))$

Then we can check that these two morphisms form an isomorphism between $(V, =)$ and $(N'_{\text{typa}}, =)$.

5.2 Typable de Bruijn Raw Terms

First we define λ -terms as an inductive family indexed by N . A_n represents λ -terms with at most n free variables with the following introduction rules [9]:

$$\frac{n : N \quad i : N'_n}{\text{var}_r(i) : A_n} \quad \frac{n : N \quad t : A_{s(n)}}{\lambda_r(t) : A_n} \quad \frac{n : N \quad s, t : A_n}{\text{app}_r(s, t) : A_n}$$

$\text{lterm} \in (N) \text{Set}$

$$\begin{aligned} \text{varI} &\in (n \in N; i \in N'(n)) \text{lterm}(n) \\ \text{lamI} &\in (n \in N; \text{lterm}(\text{succ}(n))) \text{lterm}(n) \\ \text{apI} &\in (n \in N; \text{lterm}(n); \text{lterm}(n)) \text{lterm}(n) \end{aligned}$$

Then we define a typing relation on the raw terms:

$$\Gamma \vdash t :: A = \text{lterm_typa}(\Gamma, A, t)$$

meaning that the de Bruijn term t has type A in the context Γ :

$$\frac{i :: A \in \Gamma}{\Gamma \vdash \text{var}_r(i) :: A} \quad \frac{\Gamma; A \vdash t :: B}{\Gamma \vdash \lambda_r(t) :: A \rightarrow B} \quad \frac{\Gamma \vdash t :: A \rightarrow B \quad \Gamma \vdash t' :: A}{\Gamma \vdash \text{app}_r(t, t') :: B}$$

Now we can define a family of sets $(\Sigma t : A_{|\Gamma|})(\Gamma \vdash t :: A)$, i.e. the family of typed terms indexed by contexts and types:

$\text{Typa-lterm} \in (\text{Context}; \text{Type}) \text{Set}$

$$\text{typa_lterm} \in (t \in \text{lterm}(\text{Cont-N}(\Gamma)); \text{lterm_typa}(\Gamma, A, t)) \text{Typa-lterm}(\Gamma, A)$$

where $\text{lterm_typa}(\Gamma, A, t)$ implements the typing relation $\Gamma \vdash t :: A$, and an element of the set $\text{Typa-lterm}(\Gamma, A)$ is a pair of a raw term $t \in A_{|\Gamma|}$ and a proof

that $\Gamma \vdash t :: A$. We have suppressed the arguments Γ and A in the constructor `typo_lterm`.

We now define two equalities $=_t$ and \sim on typed terms, which will correspond to the syntactic equality and $\beta\eta$ -equality on proof trees respectively.

Two typed terms of the same type are $\beta\eta$ -equal if their first parts are $\beta\eta$ -equal:

```
Eq_beet_Typa_lterm ∈ (Typa_lterm(Γ, A); Typa_lterm(Γ, A)) Set
Eq_beet_Typa_lterm(typo_lterm(a, h2), typo_lterm(a1, h)) ≡ Eq_lterm(Cont_N(Γ), a, a1)
```

where `Eq_lterm(m, a, b)` implements the $\beta\eta$ -equality on raw terms $a \sim b$:

```
Eq_lterm ∈ (lterm(n); lterm(n)) Set
ref_lterm ∈ (a ∈ lterm(n)) Eq_lterm(a, a)
sym_lterm ∈ (a, b ∈ lterm(n); Eq_lterm(a, b)) Eq_lterm(b, a)
tran_lterm ∈ (a, b, c ∈ lterm(n); Eq_lterm(a, b); Eq_lterm(b, c)) Eq_lterm(a, c)
beta_lterm ∈ (a ∈ lterm(succ(n));
              b ∈ lterm(n)
              ) Eq_lterm(apI(n, lamI(n, a), b), sub(succ(n), a, n, pair(Id_lterm(n), b)))
apcon_lterm ∈ (a1, a2 ∈ lterm(n);
               Eq_lterm(a1, a2);
               b1, b2 ∈ lterm(n);
               Eq_lterm(b1, b2)
               ) Eq_lterm(apI(n, a1, b1), apI(n, a2, b2))
sig_lterm ∈ (a, b ∈ lterm(succ(n)); Eq_lterm(a, b)) Eq_lterm(lamI(n, a), lamI(n, b))
eta_lterm ∈ (a ∈ lterm(n)) Eq_lterm(a, lamI(n, apI(succ(n), lift(n, a), varI(succ(n), 0'(n))))))
```

We may expect to define the equality on typed terms (which corresponds to the syntactic equality on proof trees) on their first part of the typed terms, i.e.

```
Eq_Typa_lterm' ∈ (Typa_lterm(Γ, A); Typa_lterm(Γ, A)) Set
Eq_Typa_lterm'(typo_lterm(a, h2), typo_lterm(a1, h)) ≡ I(a, a1)
```

However, this equality will be too coarse because it is not preserved by the map `prft` (see section 5.3), which maps a typed term in `Typa_lterm(Γ, A)` to a proof tree of the type $\Gamma \vdash A$. An example is the raw term $a = (\lambda x.y)(\lambda z.z)$. We have the following typing relation:

$$y : A \vdash a :: A$$

because $y : A \vdash \lambda x.y :: (A_1 \rightarrow A_1) \rightarrow A$ and

$$y : A \vdash \lambda z.z :: A_1 \rightarrow A_1.$$

On the other hand, we have $y : A \vdash \lambda x.y :: (A_2 \rightarrow A_2) \rightarrow A$ and

$$y : A \vdash \lambda z.z :: A_2 \rightarrow A_2$$

The map `prft` will give two different proof trees from these two different typing derivations.

Instead, the appropriate equality $=_t$ can be defined recursively in ALF as follows:

$$\begin{aligned}
 \text{Eq_syn_Typa_lterm} &\in (t_1, t_2 \in \text{Typa_lterm}(G, A)) \text{ Set} \\
 \text{Eq_syn_Typa_lterm}(\text{typa_lterm}(\text{varI}(i), h), \text{typa_lterm}(\text{varI}(i_1), h_1)) &\equiv \text{I}(i, i_1) \\
 \text{Eq_syn_Typa_lterm}(\text{typa_lterm}(\text{lamI}(h_2), \text{ex2intr}), \text{typa_lterm}(\text{lamI}(h_3), \text{ex2intr})) &\equiv \\
 \text{Eq_syn_Typa_lterm}(\text{typa_lterm}(h_2, h_5), \text{typa_lterm}(h_3, h_6)) & \\
 \text{Eq_syn_Typa_lterm}(\text{typa_lterm}(\text{aplI}(h_2, h_3), \text{ex}(a_1, _)), \text{typa_lterm}(\text{aplI}(h_4, h_5), \text{ex}(a, _))) &\equiv \\
 \text{SigmaI}(a_1, a), & \\
 [h_6]\text{And}(\text{Eq_syn_Typa_lterm}(\text{typa_lterm}(h_2, h), & \\
 \text{typa_lterm}(h_4, \text{conv_typa_lterm}(\text{I_Sub}(\text{Isym}(h_6)), h_4, h_1))), & \\
 \text{Eq_syn_Typa_lterm}(\text{typa_lterm}(h_3, h_7), & \\
 \text{typa_lterm}(h_5, \text{conv_typa_lterm}(\text{Isym}(h_6), h_5, h_8)))) &
 \end{aligned}$$

where

$$\text{conv_typa_lterm} \in (\text{I}(A, B); a \in \text{lterm}(\text{Cont-N}(G)); \text{lterm_typa}(G, A, a) \text{ lterm_typa}(G, B, a))$$

We will prove that this de Bruijn representation of the typed lambda calculus is equivalent in a strong sense to the proof tree representation described earlier in the paper. To this end we will define the de Bruijn analogues of the \mathcal{P} -cwfs $(F, =)$ and (F, \sim) and prove that they are isomorphic \mathcal{P} -cwfs. The isomorphism will be based on two maps, which will be defined in the next section, between the representation of proof trees and the representation of typed λ -calculus we just defined above.

5.3 The Map *Strip* and Its Inverse

We define a function **strip** (see [7]) which maps a proof tree t of type $\Gamma \vdash A$ to a raw term by stripping its type information. In the other direction, we define a function **prft** which maps a raw term b such that $\Gamma \vdash b :: A$ to a proof tree **prft**(b, δ) of type $\Gamma \vdash A$, where δ is a proof that $\Gamma \vdash b :: A$.

The map **strip** is defined as follows:

$$\begin{aligned}
 \text{strip} &\in (\Gamma \in \text{Context}; A \in \text{Type}; \text{Term}(\Gamma, A)) \text{lterm}(\text{Cont-N}(\Gamma)) \\
 \text{strip}(\Gamma, A, \text{Variable}(v)) &\equiv \text{varI}(\text{Var-N}(\Gamma, A, v)) \\
 \text{strip}(\Gamma, _, \text{Lambda}(b)) &\equiv \text{lamI}(\text{strip}(\text{Cons}(\Gamma, A), B, b)) \\
 \text{strip}(\Gamma, A, \text{Application}(u, v)) &\equiv \text{apl}(\text{strip}(\Gamma, \text{Arrow}(A_1, A), u), \text{strip}(\Gamma, A_1, v))
 \end{aligned}$$

It is proved that the relation $\Gamma \vdash \mathbf{strip}(t) :: A$ holds for any $t : \text{Term}(\Gamma, A)$:

$$\text{strip_Term_typa} \in (\Gamma \in \text{Context}; A \in \text{Type}; t \in \text{Term}(\Gamma, A)) \text{lterm_typa}(\Gamma, A, \text{strip}(\Gamma, A, t))$$

Therefore, we can define a family of maps **strip'**(Γ, A) : $\text{Term}(\Gamma, A) \rightarrow \text{Typa_lterm}(\Gamma, A)$:

$$\begin{aligned}
 \text{strip}' &\in (\Gamma \in \text{Context}; A \in \text{Type}; \text{Term}(\Gamma, A)) \text{Typa_lterm}(\Gamma, A) \\
 \text{strip}' &\equiv [\Gamma, A, h] \text{typa_lterm}(\text{strip}(\Gamma, A, h), \text{strip_Term_typa}(\Gamma, A, h))
 \end{aligned}$$

Theorem 7 *The map **strip** preserves the syntactic equality and $\beta\eta$ -equality on proof trees:*

$$\begin{aligned}
 \text{strip_pres_Eq_Typa_lterm} &\in (a, b \in \text{Term}(\Gamma, A); \\
 &\text{I}(a, b) \\
 &)\text{Eq_syn_Typa_lterm}(\text{strip}'(\Gamma, A, a), \text{strip}'(\Gamma, A, b))
 \end{aligned}$$

$$\begin{aligned} \text{strip_preserv_Eq} \in & (s, t \in \text{Term}(\Gamma, A); \\ & \text{EqualTerm}(s, t) \\ &) \text{Eq_lterm}(\text{Cont_N}(\Gamma), \text{strip}(\Gamma, A, s), \text{strip}(\Gamma, A, t)) \end{aligned}$$

The map $\mathbf{prft} : \text{Typa_lterm}(\Gamma, A) \rightarrow \text{Term}(\Gamma, A)$ is defined as follows:

$$\begin{aligned} \mathbf{prft} \in & (\Gamma \in \text{Context}; A \in \text{Type}; a \in \text{lterm}(\text{Cont_N}(\Gamma))); \text{lterm_typa}(\Gamma, A, a) \text{Term}(\Gamma, A) \\ \mathbf{prft}(\Gamma, A, \text{varl}(i), h) \equiv & \text{Variable}(\text{N_Var}(\Gamma, A, i, h)) \\ \mathbf{prft}(\Gamma, -, \text{laml}(h_1), \text{ex2intr}(a, b, \text{andIntr}(r(-), h_3))) \equiv & \text{Lambda}(\mathbf{prft}(\text{Cons}(\Gamma, a), b, h_1, h_3)) \\ \mathbf{prft}(\Gamma, A, \text{apI}(h_1, h_2), \text{Exists_intr}(a, \text{andIntr}(h, h_4))) \equiv & \\ & \text{Application}(\mathbf{prft}(\Gamma, \text{Arrow}(a, A), h_1, h), \mathbf{prft}(\Gamma, a, h_2, h_4)) \end{aligned}$$

Theorem 8 *The map \mathbf{prft} preserves the syntactic equality and $\beta\eta$ -equality on the set of typed terms:*

$$\begin{aligned} \mathbf{prft_pres_Eq_Typa_lterm} \in & (t_1, t_2 \in \text{Typa_lterm}(\Gamma, A); \\ & \text{Eq_syn_Typa_lterm}(t_1, t_2) \\ &) \text{I}(\mathbf{prft}'(\Gamma, A, t_1), \mathbf{prft}'(\Gamma, A, t_2)) \end{aligned}$$

$$\begin{aligned} \mathbf{prft_pres_beet} \in & (t_1, t_2 \in \text{Typa_lterm}(\Gamma, A); \\ & \text{Eq_beet_Typa_lterm}(\Gamma, A, t_1, t_2) \\ &) \text{EqualTerm}(\mathbf{prft}'(\Gamma, A, t_1), \mathbf{prft}'(\Gamma, A, t_2)) \end{aligned}$$

where \mathbf{prft}' is the uncurried version of \mathbf{prft} :

$$\begin{aligned} \mathbf{prft}' \in & (\Gamma \in \text{Context}; A \in \text{Type}; t \in \text{Typa_lterm}(\Gamma, A)) \text{Term}(\Gamma, A) \\ \mathbf{prft}'(\Gamma, A, \text{typa_lterm}(a, h)) \equiv & \mathbf{prft}(\Gamma, A, a, h) \end{aligned}$$

To prove the above theorem, we need to assume the normalisation theorem:

$$\begin{aligned} \text{Normalizing} \in & (\Gamma \in \text{Context}; A \in \text{Type}; t \in \text{Term}(\Gamma, A)) \text{Set} \\ \text{Normalizing} \equiv & [\Gamma, A, t] \text{Exists}(\text{Term}(\Gamma, A), [h] \text{And}(\text{Norm}(\Gamma, A, h), \text{EqualTerm}(t, h))) \end{aligned}$$

where $\text{Norm}(\Gamma, A, h)$ implements that the proof tree h is in normal form.

Theorem 9 *The map \mathbf{prft} is the inverse of the map \mathbf{strip} :*

$$\text{strip}'_prft' \in (a \in \text{Typa_lterm}(\Gamma, A)) \text{Eq_syn_Typa_lterm}(\text{strip}'(\Gamma, A, \mathbf{prft}'(\Gamma, A, a)), a)$$

$$\mathbf{prft}'_strip' \in (t \in \text{Term}(\Gamma, A)) \text{I}(\mathbf{prft}'(\Gamma, A, \text{strip}'(\Gamma, A, t)), t)$$

5.4 The Isomorphism of $(\mathbf{F}, =)$ $((\mathbf{F}, \sim))$ and $(\mathbf{A}_{\text{typa}}, =_t)$ $((\mathbf{A}_{\text{typa}}, \sim))$

First we extend the typing relation to tuples of raw terms:

$$\begin{aligned} \text{lterms_Typa} \in & (\Gamma, \Delta \in \text{Context}; as \in \text{Tuple}(\text{lterm}(\text{Cont_N}(\Gamma)), \text{Cont_N}(\Delta))) \text{Set} \\ \text{lterms_Typa}(\Gamma, \text{Nil}, as) \equiv & \text{One} \\ \text{lterms_Typa}(\Gamma, \text{Cons}(\Delta, A), \text{pair}(a, b)) \equiv & \text{And}(\text{lterms_Typa}(\Gamma, \Delta, a), \text{lterm_typa}(\Gamma, A, b)) \end{aligned}$$

Then we define the family of sets $(\Sigma as : A_{|\Delta|}^{\Gamma})(as :: \Gamma \rightarrow \Delta)$, which will be the set of arrows from Γ to Δ :

$$\begin{aligned} \text{Typa_lterms} \in & (\text{Context}; \text{Context}) \text{Set} \\ \text{typa_lterms} \in & (as \in \text{Tuple}(\text{lterm}(\text{Cont_N}(\Gamma)), \text{Cont_N}(\Delta))); \\ & \text{lterms_Typa}(\Gamma, \Delta, as) \\ &) \text{Typa_lterms}(\Gamma, \Delta) \end{aligned}$$

The equality on arrows are defined by the equality `Eq_syn_Typa_lterm` component-wise.

To obtain the \mathcal{P} -cwfs we need to define a few operations on `Typa_lterms(Γ , Δ)`. These operations are based on the corresponding operations on A_n (see [9]).

For example, we have the operation:

```
sub ∈ (n ∈ N; g ∈ lterm(n); m ∈ N; fs ∈ Tuple(lterm(m), n)) lterm(m)
sub(n, apl(−, h, hI), m, fs) ≡ apl(m, sub(n, h, m, fs), sub(n, hI, m, fs))
sub(n, lamI(−, h), m, fs) ≡
  lamI(m, sub(succ(n), h, succ(m), pair(map(lift(m), n, fs), varI(succ(m), 0'(m))))))
sub(n, varI(−, i), m, fs) ≡ pi(lterm(m), n, i, fs)
```

and the proof that the operation `sub` preserves typability:

```
sub_is_typa ∈ (Γ, Δ ∈ Context;
  A ∈ Type;
  bs ∈ Tuple(lterm(Cont_N(Γ)), Cont_N(Δ));
  lterms_Typa(Γ, Δ, bs);
  b ∈ lterm(Cont_N(Δ));
  lterm_typa(Δ, A, b)
) lterm_typa(Γ, A, sub(Cont_N(Δ), b, Cont_N(Γ), bs))
```

Hence, the following operation is well defined:

```
sub_Typa_lterm ∈ (Γ, Δ ∈ Context;
  A ∈ Type;
  f ∈ Typa_lterms(Γ, Δ);
  a ∈ Typa_lterm(Δ, A)
) Typa_lterm(Γ, A)
sub_Typa_lterm(Γ, Δ, A, typa_lterms(as, h), typa_lterm(aI, hI)) ≡
  typa_lterm(sub(Cont_N(Δ), aI, Cont_N(Γ), as), sub_is_typa(Γ, Δ, A, as, h, aI, hI))
```

Similarly, we can define the composition operation:

```
com_Typa_lterms ∈ (Typa_lterms(Δ, Γ); Typa_lterms(Φ, Δ)) Typa_lterms(Φ, Γ)
com_Typa_lterms(typa_lterms(as, h2), typa_lterms(asI, h)) ≡
  typa_lterms(comp(as, asI), com_Typa_lterms_typa(as, asI, h2, h))
```

where `comp` is the composition on raw terms:

```
comp ∈ (bs ∈ Tuple(lterm(n), p); as ∈ Tuple(lterm(m), n)) Tuple(lterm(m), p)
comp(bs, as) ≡ map([h]sub(n, h, m, as), p, bs)
```

and `com_Typa_lterms_typa` provides the proof object:

```
com_Typa_lterms_typa ∈ (as ∈ Tuple(lterm(Cont_N(Δ)), Cont_N(Γ));
  bs ∈ Tuple(lterm(Cont_N(Φ)), Cont_N(Δ));
  lterms_Typa(Δ, Γ, as);
  lterms_Typa(Φ, Δ, bs)
) lterms_Typa(Φ, Γ, comp(as, bs))
```

Other operations are defined similarly.

Theorem 10 ($\Lambda_{typa, =_t, \text{Type}, \text{Typa_lterm}}$) is a \mathcal{P} -cwf. Furthermore, ($\Lambda_{typa, =_t, \text{Type}, \text{Typa_lterm}}$) and ($F, =, \text{Type}, \text{Term}$) are isomorphic \mathcal{P} -cwfs.

The morphism from $\mathcal{P}\text{-cwf} (F, =)$ to $\mathcal{P}\text{-cwf} (\Lambda_{\text{typa}}, =_t)$ is the triple (H, T, \mathbf{strip}') , where the functor H on objects is the identity map and on arrows is extended by \mathbf{strip}' component-wise, and T is the identity map on Type .

The morphism from $(\Lambda_{\text{typa}}, =_t)$ to $(F, =)$ is the triple (H', T, \mathbf{prft}') , where the functor H' on objects is the identity map and on arrows is extended by \mathbf{prft}' component-wise. These morphisms form an isomorphism between $(\Lambda_{\text{typa}}, =_t)$ and $(F, =)$. These morphisms also apply to the following theorem:

Theorem 11 $(\Lambda_{\text{typa}}, \sim, \text{Type}, \text{Typa_lterm})$ is a $\mathcal{P}\text{-cwf}$ with same parts as $(\Lambda_{\text{typa}}, =_t, \text{Type}, \text{Typa_lterm})$ except the per on arrows are $\beta\eta$ -conversion \sim . Furthermore, $(\Lambda_{\text{typa}}, \sim, \text{Type}, \text{Typa_lterm})$ and $(F, \sim, \text{Type}, \text{Term})$ are isomorphic.

6 Conclusion

Many authors have proved theorems about simply and dependently typed lambda calculi inside type theory proof assistants, for example, Coquand [6], Bove [4] (in ALF), McKinna and Pollack [15], Altenkirch [1] (in LEGO), and B. Barras [3], Huet [12], Saïbi [19] (in Coq). A variety of approaches have been used. The aim of our work is not only to suggest a formalization which we claim is particularly natural from a type-theoretic point of view, but also to abstract by using category theory, and to show what is involved in showing the equivalence of different representations. This work shows that, perhaps surprisingly, it is a non-trivial problem to prove the equivalence of different representations.

What we have here is only a beginning, since we only consider and relate two rather similar representations. It would be interesting to carry out such equivalence proofs for other representations as well. It would also be very interesting to extend the present work to formalizations of dependent types. The long term goal is to get a library for proof theory which is elegant and flexible with respect to choice of representation.

Acknowledgements. I am grateful to Peter Dybjer for our many discussions, and for his encouragement, ideas, and careful reading of the drafts of this paper, and to Makoto Takeyama for his patient explanation of many notions in category theory, many discussions and careful reading of the drafts of the paper. I would like to thank Michael Hedberg and Ilya Beylin for many helpful discussions.

I am grateful for a scholarship from the Shanxi Provincial government in China.

References

- [1] T. Altenkirch. A formalisation of the strong normalisation proof for system F in LEGO. *Lecture Notes in Computer Science*, Vol. 664, pp.13–28, Springer-Verlag, 1993.

- [2] T. Altenkirch and M. Hofmann and T. Streicher. Categorical Reconstruction of a Reduction Free Normalization Proof. *Lecture Notes in Computer Science*, Vol. 953, pp. 182–199, 1995.
- [3] B. Barras. Coq en Coq. *Technical Report*, RR-3026, Inria, Institut National de Recherche en Informatique et en Automatique, 1996.
- [4] A. Bove. A Machine-assisted Proof that Well Typed Expressions Cannot Go Wrong. *Technical Report*, DCS, Chalmers University, 1998.
- [5] J. Cartmell. Generalised Algebraic Theories and Contextual Categories. *Annals of Pure and Applied Logic*, Vol. 32, pp. 209–243, 1986.
- [6] C. Coquand. From Semantics to Rules: A Machine Assisted Analysis. *Proceedings of the 7th Workshop on Computer Science Logic*, pp. 91–105, Springer-Verlag LNCS 832, 1993.
- [7] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. *Mathematical Structures in Computer Science*, Vol. 7, pp. 75–94, Feb. 1997.
- [8] D. Čubrić and P. Dybjer and P. Scott. Normalization and the Yoneda embedding. *Mathematical Structures in Computer Science*, Vol. 8, No.2, pp. 153–192, Apr. 1998.
- [9] P. Dybjer. Inductive Families. *Formal Aspects of Computing*, Vol. 6, No. 4, pp. 440–465, 1994.
- [10] P. Dybjer. Internal Type Theory. *Lecture Notes in Computer Science*, Vol. 1158, pp. 120–134, 1996.
- [11] M. Hofmann. Syntax and Semantics of Dependent Types. *Semantics of Logic of Computation*, P. Dybjer and A. Pitts, eds. Cambridge University Press, 1997.
- [12] G. Huet. Residual Theory in λ -Calculus: A Formal Development. *Journal of Functional Programming*, Vol. 4, No. 3, pp. 371–394, 1994.
- [13] B. Jacobs. Simply typed and untyped Lambda Calculus revisited. *Applications of Categories in Computer Science*, Vol. 177, pp.119–142, Cambridge University Press, 1991.
- [14] L. Magnusson and B. Nordström. The ALF Proof Editor and Its Proof Engine. *Types for Proofs and Programs*, pp. 213–237, Springer-Verlag LNCS 806, Henk Barendregt and Tobias Nipkow, eds, 1994.
- [15] J. McKinna and R. Pollack. Some Type Theory and Lambda Calculus Formalised. *Journal of Automated Reasoning*, Special Issue on Formalised Mathematical Theories, ed. F. Pfenning, 1998.
- [16] B. Nordström and K. Petersson and J. M. Smith. Programming in Martin-Löf’s type theory : An Introduction. Oxford: Clarendon, 1990.
- [17] A. Obtulowicz and A. Wiweger. Categorical, Functorial and Algebraic Aspects of the Type Free Lambda Calculus. *Universal Algebra and Applications*, pp. 399–422, Banach Center Publications 9. 1982.
- [18] A. M. Pitts. Categorical Logic. *Handbook of Logic in Computer Science*, Oxford University Press, 1997.
- [19] A. Säibi. Formalisation of a λ -Calculus with Explicit Substitutions in Coq. *Proceedings of the International Workshop on Types for Proofs and Programs*, pp. 183–202, P. Dybjer and B. Nordström and J. Smith, eds, Springer-Verlag LNCS 996, 1994.
- [20] R. A. G. Seely. Locally cartesian closed categories and type theory. *Mathematical Proceedings of the Cambridge Philosophical Society*, Vol. 95, pp. 33–48, Cambridge Philosophical Society, 1984.