Towards Formalizing Categorical Models of Type Theory in Type Theory

Alexandre Buisse[1]

Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg

Peter Dybjer[2]

Department of Computer Science and Engineering
Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg

Abstract

This note is about work in progress on the topic of "internal type theory" where we investigate the internal formalization of the categorical metatheory of constructive type theory in (an extension of) itself. The basic notion is that of a category with families, a categorical notion of model of dependent type theory. We discuss how to formalize the notion of category with families inside type theory and how to build initial categories with families. Initial categories with families will be term models which play the role of canonical syntax for dependent type theory. We also discuss the formalization of the result that categories with finite limits give rise to categories with families. This yields a type-theoretic perspective on Curien's work on "substitution up to isomorphism". Our formalization is being carried out in the proof assistant Agda 2 developed at Chalmers.

1 Introduction

Most work on the metatheory of constructive type theory use standard informal mathematical metalanguage. Although such metatheory often have an intuitive constructive character it is striking that most authors rely on classical set-theoretic notions when explaining concepts rigorously. For example, when building models of constructive type theory it is common to first build a partial interpretation function mapping raw terms to their meaning, and only afterwards show that the meaning of well-typed terms is defined. The constructive meaning of this is not obvious, at least if we use constructive type theory itself as the metalanguage. The functions in this theory are all total, so partial functions need to be encoded as total functions, and this will have formal repercussions. Another example is the treatment

[1] Email: buisse@cs.chalmers.se
[2] Email: peterd@cs.chalmers.se

This paper is electronically published in
Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

of the inductive-recursive definitions which are needed in certain model constructions and normalization proofs. Although such definitions are constructively valid [12,13,14,15] most authors rely on their interpretation in set theory [23,3,1,2] and this also has formal repercussions.

In this project we plan to show that it is possible to rely entirely on constructive notions on the metalevel. The idea of doing such constructive model theory goes back to Martin-Löf [19]. However, Martin-Löf relied on an informal constructive metalanguage, while we here are more specific and work with suitable versions of Martin-Löf type theory as formal metalanguages. We are checking our proofs in the proof assistant Agda 2 which is currently under development by Ulf Norell at Chalmers. In this way we are turning *constructive metamathematics* into *metaprogramming*. A proof of an abstract result such as "any category with finite limits is a category with families" turns into a "compiler" which maps any input data structure representing a category with finite limits into an output data structure representing a category with families. The type-system will ensure that this compiler is correct. When we program this compiler in the Agda system we can actually run it on concrete examples.

It is worth-while pointing out that Martin-Löf type theory is intended to be a full-scale language for constructive mathematics just as Zermelo-Fraenkel set theory is a full-scale language for classical mathematics. Just as it might be necessary to postulate the existence of certain additional large cardinals in order to discuss the metatheory of set theory inside set theory, we will here need to add certain analogues of large cardinals to constructive type theory.

Previous work on constructive model theory which use a formal constructive metalanguage includes Pollack's work on Lego in Lego [22] and Barras' work on Coq in Coq [4]. Both authors deal with the usual lambda calculus based syntax of constructive type theory. Here we will instead base our work on a categorical notion of model of type theory. The formal system of type theory will be represented abstractly as an *initial category with families (with extra structure)*. A category with families (cwfs) [11] is a notion of model of (the most basic rules of) dependent type theory, and the initial such is the "term model". There are several reasons for choosing a categorical approach. One of them is to achieve more "canonical" results. Syntactic approaches tend to depend on a number of detailed choices. Should we choose ordinary named variables or de Bruijn's nameless ones, or should we use de Bruijn levels? Is the rule of substitution a primitive or derived rule? Etc. Categorical notions tend to be more stable and canonical, although it must be admitted that certain representation issues remain. Other arguments for basing our approach on category theory are well-known from categorical type theory: we get a clear notion of model, access to many powerful results in category theory, and a mathematically elegant approach which hopefully also leads to a more economical formalization.

Our work builds on the second author's paper "Internal Type Theory" [11]. In that paper the notion of a category with families was introduced as a notion of model of dependent type theory with a particularly straightforward connection to syntax. The formalization of cwfs inside type theory is also discussed. Such a formalization is called an "internal type theory" since it is analogous to the notion

of *internal category*. In any category with suitable extra structure (finite limits) we can define what it means to be an internal category object. Similarly, in any cwf with extra structure (modelling Π-types, Σ-types and universes) we can define a notion of internal cwf.

Our work will rely on previous experience of formalization of category theory inside constructive type theory, see for example the development of elementary category theory in Huet and Saibi's book on Constructive Category Theory [18].

Plan of the note.

In Section 2 we present the notion of a cwf in classical metalanguage. In Section 3 we explain some of the features of the proof assistant Agda 2 which we use for our formalization. In Section 4 we present the usual approach to the formalization of categories inside type theory, continue with the formalization of the category of families of sets, and then arrive at the notion of category with families inside type theory. In Section 5 we sketch how to formalize the result that categories with finite limits give rise to cwfs. We discuss the close relationship to Curien's paper "Substitution up to Isomorphism" [7] and contrast it to a similar result by Hofmann [16] formulated using classical categorical notions. In Section 6 we discuss the construction of initial categories with families.

2 Categories with families

Categories with families (cwfs) [11,17] are variants of Cartmell's *categories with attributes* [16,21]. The point of the reformulation is to get a more direct link to the syntax of dependent types. In particular we avoid reference to pullbacks, which give rise to a conditional equation when formalized in a straightforward way. Cwfs can therefore be formalized as a generalized algebraic theory in Cartmell's sense with clear similiarities to Martin-Löf's *substitution calculus* for type theory [20].

Let **Fam** be the category of families of sets, where an *object* is a family of sets $(B(x))_{x\in A}$ and a *morphism* with source $(B(x))_{x\in A}$ and target $(B'(x'))_{x'\in A'}$ is a pair consisting of a function $f : A \to A'$ and a family of functions $g(x) : B(x) \to B'(f(x))$ indexed by $x \in A$.

The components of a cwf are named after the corresponding syntactic notions.

Definition 2.1 A *category with families* consists of the following four parts:

- A base *category* C. Its objects are called *contexts* and its morphisms are called *substitutions*.
- A *functor* $T : C^{op} \to \mathbf{Fam}$. We write $T(\Gamma) = (\Gamma \vdash A)_{A\in Type(\Gamma)}$, where Γ is an object of C, and call it the family of *terms* indexed by *types* in context Γ. Moreover, if γ is a morphism of C then the two components of $T(\gamma)$ interpret substitution in types and terms respectively. We write $A[\gamma]$ for the application of the first component to a type A and $a[\gamma]$ for the application of the second component to a term a.
- A *terminal object* $[\,]$ of C called the *empty context*.
- A *context comprehension* operation which to an object Γ of C and a type $A \in$

$Type(\Gamma)$ associates an object $\Gamma; A$ of C; a morphism $\mathrm{p}_{\Gamma,A} : \Gamma; A \to \Gamma$ of C (the *first projection*); and a term $\mathrm{q}_{\Gamma,A} \in \Gamma; A \vdash A[\mathrm{p}_{\Gamma,A}]$ (the *second projection*). The following universal property holds: for each object Δ in C, morphism $\gamma : \Delta \to \Gamma$, and term $a \in \Delta \vdash A[\gamma]$, there is a unique morphism $\theta = \langle \gamma, a \rangle : \Delta \to \Gamma; A$, such that $p \circ \theta = \gamma$ and $q[\theta] = a$.

A basic example of a cwf is obtained by letting $C = \mathrm{Set}$, the category of small sets, $Type(\Gamma)$ be the set of Γ-indexed small sets, and

$$\begin{aligned} \Gamma \vdash A &= \prod_{x \in \Gamma} A(x) \\ A[\delta](x) &= A(\delta(x)) \\ a[\delta](x) &= a(\delta(x)) \\ [\,] &= 1 \\ \Gamma; A &= \sum_{x \in \Gamma} A(x) \end{aligned}$$

Definition 2.2 Let (C, T) denote a cwf with base category C and functor T. A *morphism of cwfs* with source (C, T) and target (C', T') is a pair (F, σ), where $F : C \to C'$ is a functor and $\sigma : T \to T'F$ is a natural transformation, such that terminal object and context comprehension are preserved on the nose.

Small cwfs and morphisms of cwfs form a category **Cwf**.

As already mentioned the notion of a category with families can be formalized as a *generalized algebraic theory* in the sense of Cartmell [6]. It is instructive to look at the rules of this theory, but we do not have room in this short note to display them, and refer the reader to Dybjer [11].

3 The proof assistant Agda 2

We will work in Martin-Löf's constructive type theory and use the proof assistant Agda 2 for our formalization. Agda 2 is an implementation of Martin-Löf's logical framework with support for adding new inductively defined sets and recursively defined functions using pattern matching. It is thus suitable for implementing various fragments of Martin-Löf type theory. It can also be viewed as a dependently typed programming language. Its syntax is quite close to Haskell. The main difference to Haskell (and other standard functional languages such as OCAML) is that it has dependent types. It is important to point out that Agda does not itself force the user to define only well-founded data types and terminating functions, although at a later stage such termination and well-foundedness checkers will be available. Currently it is up to the user to make sure that a specific logical discipline is followed and to explain and justify this discipline.

If we remove Agda's dependent types, we get a language rather close to Haskell. However, Agda doesn't have the implicit Hindley-Milner polymorphism of Haskell. In Haskell we have for example the polymorphic identity function

```
id :: a -> a
```

stating that for any type `a` we have an identity funciton on it. In Agda we have to

explicitly quantify over the type Set of small types. We write

$$\mathbf{id} : (A : Set) \to A \to A$$

which means that for any small type A we have an identity function $\mathbf{id}\, A : A \to A$. (In general Agda uses the notation $(x : \alpha) \to \beta$ for the type of functions which map an object x of type α into a result in β, where the result type β may depend on x.) This is why one says that Agda is an implementation of Martin-Löf's *monomorphic* type theory: **id** has a unique type.

However, it is cumbersome to work in monomorphic type theory since one has to manipulate large expressions. Therefore Agda allows you to suppress certain arguments when they can be inferred from the context. We call such arguments *implicit*. For example, whenever the function **id** gets a second argument a, Agda tries to infer the first argument A, which is the type of a. The user can inform Agda that the first argument of **id** is such an implicit argument by enclosing it in braces:

$$\mathbf{id} : \{A : Set\} \to A \to A$$

Thus, during type-checking of an expression $\mathbf{id}\, a$ the system will know that a is the second argument and try to infer the first argument.

Sometimes a user may want to give an implicit argument explicitly, for example, in situations where the system cannot infer it. Such explicit implicit arguments are enclosed in braces. For example, in

$$\mathbf{id}\, \{Bool\} : Bool \to Bool$$

the user has made the first argument $Bool$ explicit.

Agda 2 has a notion of **record**, that is a type of tuples with named components (field). To instantiate it, one needs to instantiate all the fields. A record is really a module, and thus the field A of a record r of type R can be accessed with the syntax $R.A\ r$: declaring the record type R automatically creates a module with the same name and one projection function for each field, which take as their first argument a structure of type R.

Here is an example. A record for equivalence relations on a given set A consists of four fields: a binary relation on A together with proofs of reflexivity, symmetry, and transitivity. Formally:

$$\begin{aligned}
&\mathbf{record}\ Equivalence\ (A\ :\ Set)\ :\ Set_1\ \mathbf{where}\\
&\qquad _==_ \quad &&: A \to A \to Set\\
&\qquad refl \quad &&: \{x\ :\ A\} \to x\ ==\ x\\
&\qquad sym \quad &&: \{x\ y\ :\ A\} \to x\ ==\ y \to y\ ==\ x\\
&\qquad trans \quad &&: \{x\ y\ z\ :\ A\} \to x\ ==\ y \to y\ ==\ z \to x\ ==\ z
\end{aligned}$$

Note that the record is a *large* type, that is, a member of the universe Set_1.

The central notion of a *setoid*, that is, a set with an equivalence relation, will be used extensively:

$$\begin{aligned}
&\mathbf{record}\ Setoid\ :\ Set_1\ \mathbf{where}\\
&\qquad car \quad &&: Set\\
&\qquad rel \quad &&: Equivalence\ car
\end{aligned}$$

To get a nicer notation, we'll define a function to access directly to the carrier of a setoid:

$$|\_| : (S : Setoid) \to Set$$
$$|S| = Setoid.carS$$

4 Categories with families in type theory

We will follow the recipe for formalizing categories with families in type theory (*internal cwfs*) described in Dybjer [11]. It is well-known how to formalize basic categorical notions such as category, functor, natural transformation, etc [18] in type theory. It is also well-known how to formalize the type-theoretic analogue **Set** of the category of sets in type theory. The objects of this category are *setoids* (or *E-sets*) that is sets with equivalence relations. The arrows are functions respecting equivalence relations.

The crucial issue for the formalization of cwfs is the formalization of the category **Fam**, and we follow the approach in the paper Internal Type Theory [11]. (We will however also investigate an alternative proof-irrelevant definition of setoid-indexed families used in a recent implementation of the category of setoids in Agda 2 by Thierry Coquand and Ulf Norell.)

Once we have defined the category **Fam** it is straightforward to formalize the rest of the cwf-structure in type theory. Note that if a cwf is formally represented as a quadruple consisting of the base category, the family valued functor, the terminal object, and context comprehension, where each of these components itself is a tuple, we can "flatten" this structure into a tuple where each component corresponds to a rule in a substitution calculus for type theory. This calculus is closely related to the calculus of explicit substitutions used by Curien [7]. Like this calculus there is an explicit construction for the type conversion rule [11]. We can thus see how the type-theoretic perspective gives a rational reconstruction of Curien's calculus.

4.1 Categories

A category in type theory consists of a set of objects, hom-setoids for each pair of objects, an identity arrow for each object, composition respecting equivalence of the arrows in the hom-setoids, and proofs of the identity and associativity laws. (Note that we have explicit proof objects for each law.)

$$\begin{array}{rlll}
\textbf{open} & Setoid & & \\
\textbf{record} & Cat & : & Set_2 \ \textbf{where} \\
& Obj & : & Set_1 \\
& _ \longrightarrow _ & : & Obj \to Obj \to Setoid \\
& id & : & \{A : Obj\} \to |A \longrightarrow A| \\
& _ \circ _ & : & \{A\ B\ C : Obj\} \to |B \longrightarrow C| \to |A \longrightarrow B| \to \\
& & & \quad |A \longrightarrow C| \\
& \ldots & &
\end{array}$$

$$
\begin{aligned}
&idL \quad : \{A\ B\ :\ Obj\}\ \{f\ :\ A \longrightarrow B\} \rightarrow \_ == \_\ (rel(A \longrightarrow B))\\
&\qquad\quad (id \circ f)\ f\\
&\ldots
\end{aligned}
$$

We have chosen to formalize a notion of *locally small* category where hom-setoids must be "small" (the carrier is a set), but an object can be "large" (a member of the universe Set_1 of large sets). Examples of such locally small categories is the category **Set** of setoids and the category **Fam** of setoid-indexed families of setoids. Note also the type of locally small categories is "very large" (a member of a second universe Set_2 of very large sets).

4.2 The category ***Fam***

To define the category **Fam**, we need the notions of setoid-indexed family of setoids, and of morphism between such (the respecitve objects and arrows of **Fam**). We have seen in section 3 how to define a setoid as a record. The next step is to define a morphism between setoid as a function between the carriers together with a proof that it maps equivalent elements to equivalent elements:

$$
\begin{aligned}
&\textbf{record}\ \_ \Rightarrow \_\ (S_1\ S_2\ :\ Setoid)\ :\ Set\ \textbf{where}\\
&\qquad map\ : |S_1| \rightarrow |S_2|\\
&\qquad stab\ : \{x\ y : |S_1|\} \rightarrow x ==\ ^{3}\ y \rightarrow (map\ x) == (map\ y)
\end{aligned}
$$

Identity and composition of setoid morphisms are easy to add. For instance,

$$
\begin{aligned}
&id\ :\ \{S : Setoid\} \rightarrow S \Rightarrow S\\
&id = \textbf{record}\ \{map = \backslash x \rightarrow x\ \ ;\ stab = \backslash p \rightarrow p\}
\end{aligned}
$$

We also add an extensional equality $==_{\Rightarrow}$:

$$
\begin{aligned}
&\_ ==_{\Rightarrow} \_\ : \{S_1\ S_2 : Setoid\} \rightarrow S_1 \Rightarrow S_2 \rightarrow S_1 \Rightarrow S_2 \rightarrow Set\\
&F_1 ==_{\Rightarrow} F_2 = (\textbf{forall}\ x \rightarrow (map\ F_1\ x)\ ==_{\Rightarrow}\ (map\ F_2\ x)) \rightarrow True
\end{aligned}
$$

We are now ready to define the notion of a family of setoids indexed by a given setoid S: a fibre map that indexes setoids by elements of the carrier of the indexing setoid, a reindexing function ι that maps the equivalence relation of the indexing setoid into the indexed setoids and proofs that this reindexing function is coherent with the fact that the relation is an equivalence.

$$
\begin{aligned}
&\textbf{record}\ SetoidFam\ (S\ :\ Setoid)\ :\ Set_1\ \textbf{where}\\
&\qquad fibre \quad\ : |S| \rightarrow Setoid\\
&\qquad \iota \quad\ \ : \{x\ x' : |S|\} \rightarrow x == x' \rightarrow (fibre\ x') \Rightarrow (fibre\ x)\\
&\qquad idcoh \quad : \{x : |S|\} \rightarrow \iota\ (refl\ (rel\ S)\ \{x\}) ==_{\Rightarrow} id\ \{fibre\ x\}\\
&\qquad symcoh_L\ : \{x\ y : |S|\} \rightarrow (p : x == y) \rightarrow\\
&\qquad\qquad \iota\ (sym\ (rel\ S)\ \{x\}\ \{y\}\ p) \circ (\iota\ p) ==_{\Rightarrow} id
\end{aligned}
$$

[3] Here we redefined $\_ == \_$ with the type $\{S : Setoid\} \rightarrow Equivalence.\_ == \_\ (rel\ S)$

$$
\begin{aligned}
&\mathit{symcoh}_R \;:\; \ldots \\
&\mathit{transcoh} \;:\; \{x\ y\ z : |S|\} \to (p : x == y) \to (p' : x == z) \to \\
&\qquad \iota\ (\mathit{trans}\ (\mathit{rel}\ S)\ \{x\}\ \{y\}\ \{z\}\ p\ p') ==_{\Rightarrow} (\iota\ p) \circ (\iota\ p')
\end{aligned}
$$

The last step is to define what a morphism between objects of type $SetoidFam$ is. There is one map for each component: a map $indexmap$ between the index setoids, a map $fibremap$ between the fibres, and a proof ιmap that reindexing commutes with the map between fibres (see the upper part of the figure below):

$$
\begin{aligned}
\textbf{record}\quad \_ \Longrightarrow \_ \quad & \{S_1\ S_2 :\ \mathit{Setoid}\} \\
& (F_1 : \mathit{SetoidFam}\ S_1)\ (F_2 : \mathit{SetoidFam}\ S_2) :\ \mathit{Set}_1\ \textbf{where} \\
\mathit{indexmap} &: S_1 \Rightarrow S_2 \\
\mathit{fibremap} &: (x : |S_1|) \to (\mathit{fibre}\ F_1\ x) \Rightarrow (\mathit{fibre}\ F_2\ ((\mathit{map}\ \mathit{indexmap}\ x)) \\
\iota map \quad &: \{x\ x' : |S_1|\} \to (p : x == x') \to ((\mathit{fibremap}\ x) \circ (\iota\ F_1\ p)) \\
& ==_{\Rightarrow} (\iota\ F_2\ (\mathit{stab}\ \mathit{indexmap}\ p)) \circ (\mathit{fibremap}\ x')
\end{aligned}
$$

The figure 1 illustrates the structure of a morphism in **Fam**.

The rest of the properties needed for **Fam** to be a category are then straightforward, though particularily tedious.

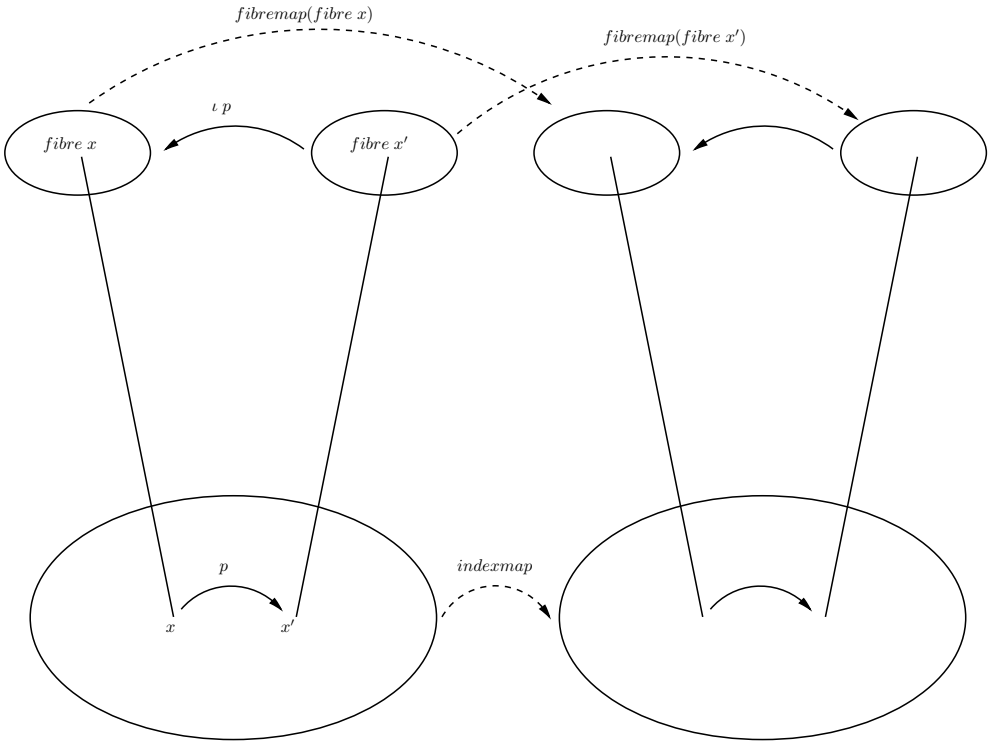

Fig. 1. Representation of $F_1 \Longrightarrow F_2$

4.3 Cwfs

Just as in the classical definition of a cwf in Section 2, a cwf inside type theory is a quadruple consisting of a base category C, a functor $T : C^{op} \to \mathbf{Fam}$, a terminal object of C, and a context comprehension. Above we have outlined the type theoretic definition of a category and of the category **Fam**. The type-theoretic definition of a functor is well-known [18]. Furthermore, it is clear what a terminal object is type-theoretically, and we can express the structure of context comprehension type-theoretically. All this leads to the definition of cwf inside type theory (i.e. the notion of internal cwf), although we do not have room to display the details.

We remark that our formalization of locally small cwfs in type theory has only used a logical framework with Π-types, records (we could equivalently use Σ-types), and the universes Set, Set_1, and Set_2. (We don't need Set_2 if we only want to formalize small cwfs.)

4.4 Cwfs with extra structure

The notion of cwf just models the most basic structure of dependent types: context and variable formation and substitution in types and terms. Therefore we usually want to work with cwfs with extra structure corresponding to adding type formers (Π, Σ, universes, natural numbers, etc) to dependent type theory. This does not give rise to any further formalization problems. See the Internal Type Theory paper [11] for further explanation.

5 Categories with finite limits are cwfs

Here we outline the proof inside type theory that categories with finite limits are cwfs. This proof will help us understand how cwfs relate to standard ideas in categorical type theory: why types can be modelled as projection arrows, why terms can be modelled as sections of these projections, and why substitution in types can be modelled by pullbacks. We will discuss the type-theoretic perspective on the problem of "substitution-up-to-isomorphism" and show the similarity with Curien's approach [7]. We will also contrast it to Hofmann [16], who used standard categorical notions assuming set-theoretic metalanguage.

5.1 Categories with finite limits in type theory

Categories with finite limits can be formalized as categories with terminal objects and pullback. (A category with terminal objects and pullbacks has all finite limits.) As a type-theoretic structure a pullback is a function that given three objects and two arrows constructs an object, two arrows, proofs of commutativity of the square, and a proof of the universal property. Here is the formalization in Agda.

$$
\begin{array}{rl}
\textbf{record} \quad IsPull & \{A\ B\ C\ D : Obj\}(f : A \longrightarrow B)(g : A \longrightarrow C)(f' : C \longrightarrow D) \\
& (g' : B \longrightarrow D)(square : g' \circ f == f' \circ g) : Set_1\ \textbf{where} \\
h \quad : (A' : & Obj)(h_1 : A' \longrightarrow C)(h_2 : A' \longrightarrow B)(tr : f' \circ h_1 == g' \circ h_2) \\
& \to \exists!\ \backslash(h : A' \longrightarrow A) \to (g \circ h == h_1) \wedge (f \circ h == h_2)
\end{array}
$$

$$\begin{array}{l}\textbf{record}\ \mathit{Pullback}\,\{B\ C\ D : \mathit{Obj}\}(g' : B \longrightarrow D)(f' : C \longrightarrow D) : \mathit{Set}_1\ \textbf{where}\\ \quad\begin{array}{rl} A & : \mathit{Obj}\\ f & : A \longrightarrow B\\ g & : A \longrightarrow C\\ \mathit{square} & : g' \circ f == f' \circ g\\ \mathit{pull} & : \mathit{isPull}\ f\ g\ f'\ g'\ \mathit{square}\end{array}\end{array}$$

$$\begin{array}{l}\textbf{record}\ \mathit{PullCat} : \mathit{Set}_2\ \textbf{where}\\ \quad \mathit{pullprop} : \{B\ C\ D : \mathit{Obj}\}(g' : B \longrightarrow D)(f' : C \longrightarrow D) \to \mathit{pullback}\ g'\ f'\end{array}$$

5.2 Slice categories

We shall recover the structure of cwfs by modelling types by objects in slice categories, and by modelling substitution in types by (the object part of) the pullback functor between slice categories.

Given any category C and an object Γ of that category we can construct the slice category C/Γ. The objects are pairs of objects A in C and arrows $f : A \longrightarrow C$. The proof that C/Γ is a category is quite easily derived from the fact that C is also a category.

5.3 Cwfs from categories with finite limits

We get the cwf structure from a category with finite limits in the following way:

- The base categories are the same.
- The set of types in a context Γ is the set of objects of the slice category C/Γ. Equality of types is isomorphism in the slice category.
- The set of terms of a given type A in context Γ is the set of sections of the arrow in C with target Γ which models A. Equality of terms is inherited from equality of arrows in the base category. (Proofs that these arrows are sections are not relevant to the equality.)
- Substitution in types is obtained by the pullback construction.
- Substitution in terms is also extracted from the pullback.
- Etc.

We here show part of the Agda formalization of how substitution in types is modelled by the pullback construction (types are omitted when not important).

An object of the slice category C/Γ is an arrow, but in order to typecheck, we need to also specify the codomain of this arrow:

$$\begin{array}{l}\textbf{record}\ \mathit{SlObj}\,(\Gamma : \mathit{Obj}) : \mathit{Set}_1\ \textbf{where}\\ \quad\begin{array}{rl} \mathit{dom} & : \mathit{Obj}\\ \mathit{arr} & : \mathit{dom} \longrightarrow \Gamma\end{array}\end{array}$$

A section of an arrow $f : \Delta \to \Gamma$ is

$$\begin{aligned}&\textbf{record}\ \mathit{Section}\ (\Gamma : \mathit{Obj})\ (A : \mathit{SlObj}\ \Gamma) : \mathit{Set}\ \textbf{where}\\ &\qquad \mathit{sect} \quad : \Gamma \longrightarrow (\mathit{dom}\ A)\\ &\qquad \mathit{id}_L \quad : (\mathit{arr}\ A) \circ \mathit{sect} == \mathit{id}\end{aligned}$$

We are now ready to proceed:

$$\begin{aligned}&\mathit{Context} = \mathit{Obj}\\ &\mathit{Subst}\ \Gamma\ \Delta = \Gamma \longrightarrow \Delta\\ &\mathit{Type}\ \Gamma = \mathit{SlObj}\ \Gamma\\ &\mathit{Term}\ \Gamma\ A = \mathit{Section}\ \Gamma\ A\end{aligned}$$

$$\begin{aligned}&\mathit{subst}\ : \{\Gamma\ \Delta :\ \mathit{Context}\} \to \mathit{Type}\ \Gamma \to \mathit{Subst}\ \Delta\ \Gamma \to \mathit{Type}\ \Delta\\ &\mathit{subst}\ \{\Gamma\}\ \{\Delta\}\ T\ g = \textbf{let}\ p = \mathit{pullprop}\ (\mathit{arr}\ \Gamma\ T)\ g\ \textbf{in}\\ &\qquad \mathit{record}\ \ \{\mathit{dom} = \mathit{pullback}.A\ p;\ \mathit{arr} = \mathit{pullback}.g\ p\}\end{aligned}$$

We can compare our interpretation to the approach taken in the paper "Substitution up to Isomorphism" by Curien [7]. Like us, Curien interprets equality of types as isomorphism in the slice category. Another similarity is that he uses an explicit substitution calculus for dependent type theory not unlike our initial cwf which has an explicit constructor for applications of the rule of type equality.

This approach can be contrasted to Hofmann's work on interpreting type theory in locally cartesian closed categories [16]. In this work he shows how to construct a category with attributes from a category with finite limits using a technique due to Bénabou [5]. Since categories with attributes are equivalent to categories with families this ought to be highly relevant to our work. However, Hofmann uses standard category theory relying on set-theoretic metalanguage, and his notion of category with attributes is a "strict" one, just as our set-theoretic notion of cwf in Section 2. To show that in this classical setting categories with finite limits form cwfs, we cannot just interpret substitution as "chosen" pullbacks, unless this choice satisfies the laws of substitution in types *up to equality*. Hofmann states that it is an open problem to find such a choice. When working in type-theoretic metalanguage on the other hand we have the freedom to interpret equality of types as isomorphism of objects, and thus there is no need for Bénabou's construction.

However, what we gain when avoiding Bénabou's construction we have to pay back when constructing cwfs (with extra structure) from syntax and proving their initiality. This work is similar to the *coherence problem* discussed by Curien.

6 Initial cwfs (with extra structure)

If we work in set-theoretic metalanguage initial cwfs exist. This is a corollary of a theorem of Cartmell [6] who showed that any generalized algebraic theory has an initial model in an appropriate categorical sense.

We shall discuss two ways of constructing initial cwfs with extra structure inside type theory. Without the extra structure the initial cwf is trivial; it is nothing but the category with one object and one arrow, where the family valued functor returns the empty family of sets. To get interesting extra structure we postulate the

existence of Π and Σ and a universe of small types. We call this an *LF-cwf*, since it is the categorical analogue of Martin-Löf's logical framework. We remark that the discussion below is not very dependent on the exact choice of extra structure, except that some properties will rely on normalization.

6.1 Strongly typed version

This version is obtained by taking the definition of an LF-cwf as a record and turn it into an inductive definition. The notion of LF-cwf specifies seven different families of sets, one corresponding to each of the seven forms of judgement. Each of these will turn into a formation rule for seven inductively defined sets of "derivations" of judgements. The notion of LF-cwf furthermore specifies a number of different operations each corresponding to a rule of inference. Each of these operations will become a constructor in the inductive definition of the initial cwf.

For instance, contexts are defined by the grammar $\Gamma ::= [\,] \mid \Gamma \triangleright A$ where A is a type. Correspondingly there will be two constructors for contexts in the initial cwf, with the following types formalized in Agda:

```
mutual
  data Ctxt : Set where
    [] : Ctxt
    ▷ : (Γ : Ctxt) → Type Γ → Ctxt
  data Type : Ctxt → Set where
    ...
```

However, it is important to remark that this inductive definition falls outside the standard schema of mutual inductive definitions in constructive type theory [10]. Nevertheless, we believe that it is a constructively meaningful definition. As part of our investigation we plan to generalize the schema in [10,15] to cover that schema, and also to provide set-theoretic semantics by extending [9].

6.2 The category of cwfs in type theory.

Although the above seems like a reasonable candidate for a strongly typed notion of *term model* of type theory, we would like to prove formally in type theory that we have an initial LF-cwf, that is, that it forms an initial object in the category of LF-cwfs. In Section 2 we defined a notion of cwf morphism which preserves chosen structure "on the nose". However,the type-theoretic definition of a category does not equip objects with a notion of equality. The natural notion of equality of objects is isomorphism, and hence we would like to use a notion of cwf morphism which preserves the cwf structure up to isomorphism. To spell out the definition of the category of cwfs and construct an initial object (together with the unique arrow to another object) is another part of our project. Given such a definition it should be straightforward to see that the above strongly typed version is initial since it means that each construction is interpreted as the corresponding notion in a given cwf. In a sense the elimination principle *is* the unique arrow from the initial cwf to an arbitrary cwf, at least roughly speaking, cf e g the proof of the correspondence between initiality and elimination principles in [14].

6.3 Raw term version

An alternative definition of the initial cwf can be obtained by first defining raw contexts, raw types, etc.

mutual

data $RawCtxt : Set$ **where**

$[\,] : RawCtxt$

$\triangleright : RawCtxt \to RawType \to RawCtxt$

data $RawType : Set$ **where**

$\ldots$

As a second step we define a predicate "valid context" on $RawCtxt$, a binary relation "valid type" between $RawCtxt$ and $RawType$, etc. In this way we give a mutual inductive definition of all the seven forms of judgement viewed as predicates on raw notions.

Finally, we would like to show that this also yields an initial cwf by defining a cwf-structure-preserving map into an arbitary cwf, and to show the uniqueness of this map.

7 Conclusion and future work

As already mentioned this is work in progress. An auxiliary aim is to test the suitability of the new proof assistant Agda 2 for the purpose of formalizing category theory. Agda 2.0.0 was just released (June 2007) and still lacks many features of a more mature system such as Coq or even its predecessors AgdaLight, Agda 1 and Alfa. For example, there is still no support for equational reasoning and automatic proof construction. The implicit argument feature of Agda 2 is used heavily in this work, but we have encountered some performance problems.

After completing the formalizations described in this paper, we would like to add more structure to categories with families. In particular we would like to formalize the full Seely-Curien [24,7] interpretation of Martin-Löf type theory (understood as categories with families with extra structure modelling Π- and Σ-types and extensional equality types) in locally cartesian closed categories.

Another direction of future research would be to formalize key metatheoretical results of Martin-Löf type theory such as decidability of equality and type-checking based on categories with families [1,2]. This is related to the work by Danielsson [8] who presented such a formalization of a normalization by evaluation result in the system AgdaLight, a precursor of the Agda 2 system. Danielsson did however not base his work on a categorical presentation of dependent type theory.

References

[1] A. Abel, K. Aehlig, and P. Dybjer. Normalization by evaluation for Martin-Löf type theory with one universe. In *"23rd Conference on the Mathematical Foundations of Programming Semantics, MFPS XXIII*, Electronic Notes in Theoretical Computer Science, pages 17–40. Elsevier, 2007.

[2] A. Abel, T. Coquand, and P. Dybjer. Normalization by evaluation for Martin-Lf type theory with equality judgements. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science*, July 2007. To appear.

[3] P. Aczel. *Frege Structures and the Notions of Proposition, Truth, and Set*, pages 31–59. North-Holland, 1980.

[4] B. Barras. *Auto-validation d'un système de preuves avec familles inductives*. Thèse de doctorat, Université Paris 7, Nov. 1999.

[5] J. Benabou. Fibered categories and the foundations of naive category theory. *J. Symb. Log*, 50(1):10–37, 1985.

[6] J. Cartmell. Generalized algebraic theories and contextual categories. *Annals of Pure and Applied Logic*, 32:209–243, 1986.

[7] P.-L. Curien. Substitution up to isomorphism. *Fundamenta Informaticae*, 19(1,2):51–86, 1993.

[8] N. A. Danielsson. A formalisation of a dependently typed language as an inductive-recursive family. In *Proceedings of the TYPES meeting 2006*. Springer-Verlag, 2007.

[9] P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 280–306. Cambridge University Press, 1991.

[10] P. Dybjer. Inductive families. *Formal Aspects of Computing*, 6:440–465, 1994.

[11] P. Dybjer. Internal type theory. In *TYPES '95, Types for Proofs and Programs*, number 1158 in Lecture Notes in Computer Science, pages 120–134. Springer, 1996.

[12] P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525–549, June 2000.

[13] P. Dybjer and A. Setzer. A finite axiomatization of inductive-recursive definitions. In J.-Y. Girard, editor, *Typed Lambda Calculi and Applications*, volume 1581 of *Lecture Notes in Computer Science*, pages 129–146. Springer, April 1999.

[14] P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1–47, 2003.

[15] P. Dybjer and A. Setzer. Indexed induction-recursion. *Journal of Logic and Algebraic Programming*, 2006.

[16] M. Hofmann. On the interpretation of type theory in locally cartesian closed categories. In L. Pacholski and J. Tiuryn, editors, *CSL*, volume 933 of *Lecture Notes in Computer Science*. Springer, 1994.

[17] M. Hofmann. Syntax and semantics of dependent types. In A. Pitts and P. Dybjer, editors, *Semantics and Logics of Computation*. Cambridge University Press, 1996. To appear.

[18] G. Huet and A. Saibi. Constructive category theory. In *Proceedings of the Joint CLICS-TYPES Workshop on Categories and Type Theory, Göteborg*, January 1995.

[19] P. Martin-Löf. About models for intuitionistic type theories and the notion of definitional equality. In S. Kanger, editor, *Proceedings of the 3rd Scandinavian Logic Symposium*, pages 81–109, 1975.

[20] P. Martin-Löf. Substitution calculus. Notes from a lecture given in Göteborg, November 1992.

[21] A. M. Pitts. Categorical logic. In *Handbook of Logic in Computer Science*. Oxford University Press, 1997. Draft version of article to appear.

[22] R. Pollack. *The Theory of Lego A Proof Checker for the Extended Calculus of Constructions*. PhD thesis, University of Edinburgh, 1994.

[23] D. S. Scott. Combinators and classes. In C. Böhm, editor, *Lambda-Calculus and Computer Science Theory*, volume 37, pages 1–26, 1975.

[24] R. A. G. Seely. Locally cartesian closed categories and type theory. *Proceedings of the Cambridge Philosophical Society*, 95:33–48, 1984.