On the Algebraic Foundation of Proof Assistants for Intuitionistic Type Theory

Andreas Abel[1], Thierry Coquand[2], and Peter Dybjer[2]

[1] Institut für Informatik, Ludwig-Maximilians-Universität
Oettingenstr. 67, D-80538 München
[2] Department of Computer Science, Chalmers University of Technology
Rännvägen 6, S-41296 Göteborg

Abstract. An algebraic presentation of Martin-Löf's intuitionistic type theory is given which is based on the notion of a category with families with extra structure. We then present a type-checking algorithm for the normal forms of this theory, and sketch how it gives rise to an initial category with families with extra structure. In this way we obtain a purely algebraic formulation of the correctness of the type-checking algorithm which provides the core of proof assistants for intuitionistic type theory.

1 Introduction

The type-checking algorithm [6] is the core of proof assistants for intensional dependent type theories such as Coq [3], Agda [13], and Epigram [5]. Such a proof assistant is essentially a tool for checking whether a given term a has a given type A relative to a context Γ:

$$\Gamma \vdash a : A$$

The user writes a type A representing a proposition to be proved, and the proof assistant aids her in constructing a proof a which witnesses the truth of A.

We shall here assume that Γ, A, and a are all in normal form with respect to the reduction rules, although this restriction may not be strictly imposed in proof assistants.

In this note we shall present a new algebraic approach to the correctness of the type-checking algorithms. Such correctness is not only important for the trust in the proof assistants, it is also philosophically significant. The decidability of typing is one of the main reasons for preferring intensional [10,12] to extensional type theory [9]. According to a certain point of view in constructivism it should be mechanically decidable whether a certain construction a is a witness to the truth of a given proposition A.

We will here consider a core dependent type theory: Martin-Löf's intuitionistic type theory where the only type formers are dependent function types and a universe of small types. This is essentially Martin-Löf's *logical framework* [10,12],

J. Garrigue and M. Hermenegildo (Eds.): FLOPS 2008, LNCS 4989, pp. 3–13, 2008.

except that we here consider β-conversion only and do not have the η-rule. Moreover, we use the same normal terms for codes for small types and for the small types themselves. In this sense our universe of small type is formulated à la Russell, in spite of the fact that our algebraic framework inevitably uses universes à la Tarski.

We expect our approach to extend smoothly if we add more type formers such as $\Sigma, +, \mathrm{N}, \mathrm{N}_n$ to our theory. We also expect that our approach can be extended to deal with η-conversion [1,2].

Martin-Löf type theory is usually expressed as a system of axioms and inference rules with four forms of judgements

$$\begin{aligned}
&\Gamma \vdash a : A \\
&\Gamma \vdash A \ \mathit{type} \\
&\Gamma \vdash a = a' : A \\
&\Gamma \vdash A = A'
\end{aligned}$$

Implicitly, there is also a judgement expressing the correctness of contexts:

$$\Gamma \vdash$$

In this inference rule presentation it is not assumed that contexts, types, and terms are normal. However, we expect that whenever Γ, A, and a are normal (with respect to the reduction rules of the theory) then the type-checking algorithm will accept $\Gamma \vdash a : A$ whenever it is a provable from the axioms and inference rules of Martin-Löf type theory.

The situation is analogous for the judgements $\Gamma \vdash A \ \mathit{type}$, and $\Gamma \vdash$, although the proof assistant may not give the user access to them.

As regards the equality judgements, we have that if Γ, A, a, and a' are normal, then $\Gamma \vdash a = a' : A$ is derivable by the axioms and inference rules iff a and a' are identical (up to α-congruence), and similarly for $\Gamma \vdash A = A'$.

However, in spite of many years of research into type-checking dependent types a completely satisfactory state of affairs has not yet been reached. On the one hand it has shown difficult to use traditional methods to obtain a clear proof of some essentially lemmas, such as the fact that the dependent function space former Π is one-to-one. On the other hand there are many different syntactic formulations of dependent type theory, and it is not clear which is the canonical one. There are different treatments of variables. Should we use explicit or implicit substitutions? Is the inference rule for substitution primitive? Should we use Curry or Church-style lambda terms? Etc.

In this note we present an algebraic formulation of Martin-Löf's intensional intuitionistic type theory which is based on the notion of a *category with families (cwfs)* [7]. In this way we hope to achieve a more satisfactory basis for developing the metatheory of type theory.

There are several reasons for prefering an algebraic formulation to the usual formulations based on the lambda calculus:

– It can be argued that it is more "canonical". There is less freedom of choice of syntactic detail.
– The presentation becomes cleaner since we do not first need to prove a number of meta-theorems of syntax.
– You get a clearer notion of model which is easier to work with.

We shall here present a type-checking algorithm inspired by the notion of categories with families. We have here also benefited from our recent work on *normalization by evaluation (nbe)* [1,2]. In these papers decidability of equality is proved for some fairly standard lambda-calculus based formulations of Martin-Löf type theory. We propose to extend this work and also formulate nbe for categories with families (with extra structure).

The rest of the note is organized as follows. We first recall the notion of a category with families. Then we extend this notion with extra structure for interpreting dependent function types and universes. Finally, we outline how to construct the cwf of type-checked normal forms.

2 Categories with Families

Categories with families (cwfs) [7,8] is a categorical notion of model of the most basic rules of dependent type theory; those which deal with context formation, variables, and substitution. Categories with families are equivalent to Cartmell's categories with attributes, but the reformulation makes it possible to obtain a straightforward correspondence to the inference rules of dependent type theory, especially when formulated as a calculus of explicit substitutions, see Martin-Löf [11].

A category with families consists of a category C and a family-valued functor $T : C^{op} \to \mathbf{Fam}$, where C has a terminal object. Moreover, there is an operation of *context comprehension* closely related to Lawvere's notion of comprehension for hyperdoctrines.

Here **Fam** is 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 $x \in A$.

C is the category of contexts and substitutions. If $\Gamma \in |C|$ is a context, then $T(\Gamma)$ is the family of terms of a type A in Γ which is indexed by the well-formed types A in Γ. The arrow part of the functor T represents substitution in types and terms. The terminal object of C represents the empty context and the terminal arrow represents the empty substitution. The context comprehension operation provides representations for context extension, substitution extension, assumption, and a weakening substitution. The reader is referred to Dybjer [7] and Hofmann [8] for details.

The category **Cwf** is obtained by defining the notion of cwf-morphism as follows. 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.

The notion of a category with families can be formalized as a *generalized algebraic theory* in the sense of Cartmell [4]. Generalized algebraic theories generalize many-sorted algebraic theories, by using dependent types. They consist of *sort symbols, operator symbols,* and *equations* between well-formed *sort expressions.* Here we present the generalized algebraic theory of categories with families using inference rule notation, to highlight the fact that it provides a variable-free substitution calculus for dependent types. To improve readability we use "polymorphic" notation. For example, we write $\delta \circ \gamma$ instead of the proper $\delta \circ_{\Theta,\Delta,\Gamma} \gamma$, etc.

Rules for the category $\mathcal{C}$

$$\text{Ctxt sort}$$

$$\frac{\Delta, \Gamma : \text{Ctxt}}{\Delta \to \Gamma \text{ sort}}$$

$$\frac{\Theta, \Delta, \Gamma : \text{Ctxt} \qquad \gamma : \Delta \to \Gamma \qquad \delta : \Theta \to \Delta}{\gamma \circ \delta : \Theta \to \Gamma}$$

$$\frac{\Gamma : \text{Ctxt}}{\text{id}_\Gamma : \Gamma \to \Gamma}$$

$$\begin{aligned} (\gamma \circ \delta) \circ \theta &= \gamma \circ (\delta \circ \theta) \\ \text{id}_\Gamma \circ \gamma &= \gamma \\ \gamma \circ \text{id}_\Gamma &= \gamma \end{aligned}$$

Rules for the functor $\mathcal{T}$

$$\frac{\Gamma : \text{Ctxt}}{\text{Ty}(\Gamma) \text{ sort}}$$

$$\frac{\Gamma : \text{Ctxt} \qquad A : \text{Ty}(\Gamma)}{\Gamma \vdash A \text{ sort}}$$

$$\frac{\Delta, \Gamma : \text{Ctxt} \qquad A : \text{Ty}(\Gamma) \qquad \gamma : \Delta \to \Gamma}{A[\gamma] : \text{Ty}(\Delta)}$$

$$\frac{\Delta, \Gamma : \text{Ctxt} \quad A : \text{Ty}(\Gamma) \quad a : \Gamma \vdash A \quad \gamma : \Delta \to \Gamma}{a[\gamma] : \Delta \vdash A[\gamma]}$$

$$\begin{aligned} A[\gamma \circ \delta] &= A[\gamma][\delta] \\ A[\text{id}_\Gamma] &= A \\ a[\gamma \circ \delta] &= a[\gamma][\delta] \\ a[\text{id}_\Gamma] &= a \end{aligned}$$

Rules for the terminal object

$$[\,] : \mathrm{Ctxt}$$

$$\frac{\Gamma : \mathrm{Ctxt}}{\langle\rangle_\Gamma : \Gamma \to [\,]}$$

$$\langle\rangle_\Gamma \circ \gamma = \langle\rangle_\Gamma$$
$$\mathrm{id}_{[\,]} = \langle\rangle_{[\,]}$$

Rules for context comprehension

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\Gamma; A : \mathrm{Ctxt}}$$

$$\frac{\Delta, \Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad \gamma : \Delta \to \Gamma \quad a : \Delta \vdash A[\gamma]}{\langle \gamma, a \rangle : \Delta \to \Gamma; A}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\mathrm{p}_{\Gamma,A} : \Gamma; A \to \Gamma}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma)}{\mathrm{q}_{\Gamma,A} : \Gamma; A \vdash A[\mathrm{p}_{\Gamma,A}]}$$

$$\mathrm{p}_{\Gamma,A} \circ \langle \gamma, a \rangle = \gamma$$
$$\mathrm{q}_{\Gamma,A}[\langle \gamma, a \rangle] = a$$
$$\langle \delta, a \rangle \circ \gamma = \langle \delta \circ \gamma, a[\gamma] \rangle$$
$$\mathrm{id}_{\Gamma;A} = \langle \mathrm{p}_{\Gamma,A}, \mathrm{q}_{\Gamma,A} \rangle$$

3 Adding Dependent Function Types and a Universe of Small Types

Categories with families only provide the most basic structure for interpreting dependent type theories, and provide no structure for interpreting any type formers at all. In these notes we consider a type theory with dependent function types and one universe. To interpret these we need some extra structure. We present this structure by adding new operators corresponding to the formation, introduction, and elimination rules for the new type constructor, and to add new equations corresponding to the equality rules. This is done by translating the usual inference rules of type theory into the variable free language of categories with families.

Rules for dependent function types

$$\frac{\Gamma : \mathrm{Ctxt} \qquad A : \mathrm{Ty}(\Gamma) \qquad B : \mathrm{Ty}(\Gamma; A)}{\Pi(A, B) : \mathrm{Ty}(\Gamma)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad B : \mathrm{Ty}(\Gamma; A) \quad b : \Gamma; A \vdash B}{\lambda(b) : \Gamma \vdash \Pi(A, B)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \quad A : \mathrm{Ty}(\Gamma) \quad B : \mathrm{Ty}(\Gamma; A) \quad c : \Gamma \vdash \Pi(A, B) \quad a : \Gamma \vdash A}{\mathrm{ap}(c, a) : \Gamma \vdash B\,[\langle \mathrm{id}_\Gamma, a\rangle]}$$

$$\begin{aligned}
\Pi(A, B)\,[\gamma] &= \Pi(A\,[\gamma], B\,[\langle \gamma \circ \mathrm{p}_{\Gamma,A}, \mathrm{q}_{\Gamma,A}\rangle]) \\
\lambda(b)\,[\gamma] &= \lambda(b\,[\langle \gamma \circ \mathrm{p}_{\Gamma,A}, \mathrm{q}_{\Gamma,A}\rangle]) \\
\mathrm{ap}(c, a)\,[\gamma] &= \mathrm{ap}(c\,[\gamma], a\,[\gamma]) \\
\mathrm{ap}(\lambda(b), a) &= b\,[\langle \mathrm{id}_\Gamma, a\rangle]
\end{aligned}$$

The three first of the five equations represent the laws for substitution under Π, λ, and ap. The fourth represents β-conversion.

Rules for a universe of small types

$$\frac{\Gamma : \mathrm{Ctxt}}{\mathrm{U} : \mathrm{Ty}(\Gamma)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad a : \Gamma \vdash \mathrm{U}}{\mathrm{T}(a) : \mathrm{Ty}(\Gamma)}$$

$$\frac{\Gamma : \mathrm{Ctxt} \qquad a : \Gamma \vdash \mathrm{U} \qquad b : \Gamma; \mathrm{T}(a) \vdash \mathrm{U}}{\hat{\Pi}(a, b) : \mathrm{U}}$$

$$\begin{aligned}
\mathrm{U}\,[\gamma] &= \mathrm{U} \\
\mathrm{T}(a)\,[\gamma] &= \mathrm{T}(a\,[\gamma]) \\
\hat{\Pi}(a, b)\,[\gamma] &= \hat{\Pi}(a\,[\gamma], b\,[\langle \gamma \circ \mathrm{p}_{\Gamma,A}, \mathrm{q}_{\Gamma,A}\rangle]) \\
\mathrm{T}(\hat{\Pi}(a, b)) &= \Pi(\mathrm{T}(a), \mathrm{T}(b))
\end{aligned}$$

This is a universe of small types which is closed under dependent function types. This formulation is inevitably à la Tarski rather than à la Russell.

A cwf with extra structure for dependent function types and a universe will be called a ΠU-cwf . We can extend the notion of cwf-morphism to a notion of morphism of ΠU-cwfs by requiring that all extra structure is preserved on the nose. Let $\mathbf{Cwf}_{\mathbf{\Pi U}}$ be the category of ΠU-cwfs and ΠU-cwf -morphisms. Since ΠU-cwfs can be described as a generalized algebraic theory, it follows from a general result by Cartmell that $\mathbf{Cwf}_{\mathbf{\Pi U}}$ has an initial object, given syntactically by derivations in a certain formal system for generalized algebraic theories. This initial object is the "syntax-free" representation of a version of Martin-Löf type theory.

4 A ΠU-cwf of Normal Forms

We shall now suggest how to build the ΠU-cwf $\mathcal{N}$ of type-checked normal forms. We write some Haskell code and explain how to define $\mathcal{N}$ in terms of it. We would like to emphasize that the content of this section is preliminary. We have not yet proved our type-checking algorithm correct.

First, we introduce raw syntax for normal terms t (including normal types). They are generated together with the auxiliary subclass of neutral terms s:

$$t ::= s \mid \lambda(a) \mid \Pi(a, a) \mid \mathrm{U}$$
$$s ::= i \mid \mathrm{ap}(s, t)$$

where i is a natural number (a de Bruijn index). Raw normal contexts and raw normal substitutions are represented as lists of normal terms.

Note that these raw normal terms are not type-decorated! This is unlike the notation for cwfs, where contexts and type-arguments are part of the official notation but were sometimes surpressed to improve readability.

The category $\mathcal{N}$ will be built up by type-checked normal forms. We could write the type-checking algorithm in Haskell by introducing the data types of normal and neutral expressions defined as follows:

```
data No = Ne Ne | Lam No | Pi No No | U
data Ne = Var Int | App Ne No
```

However, for simplicity we will define the type-checking algorithm on the type of all (raw) expressions

```
data Exp = Var Int | App Exp Exp | Lam Exp | Pi Exp Exp | U
```

although it is intended to be applied only to those expressions in `Exp` which are normal.

To this end we define four functions; `isCo, isSu, isTy,` and `isTm` which will check the correctness of contexts, substitutions, types, and terms, respectively. Here a type is represented by a raw expression, and substitutions and contexts by lists of raw expressions:

```
type Ty    = Exp
type Subst = [Exp]
type Cxt   = [Ty]
```

Checking contexts

```
isCo :: Cxt -> Bool
isCo []      = True
isCo (a:cxt) = isCo cxt && isTy cxt a
```

checks whether a list of expressions represents a correct context. Such lists of expressions will be the objects in the category of contexts of $\mathcal{N}$.

Checking substitutions

```
isSu :: Cxt -> Cxt -> Subst -> Bool
isSu cxt []     []     = True
isSu cxt (b:bs) (t:ts) = isSu cxt bs ts &&
                         isTm cxt (subst b cxt) t
```

checks whether a list of expressions (the third argument) is a correct substitution with respect to a source and a target context (the first and second argument). Such substitutions wil be the arrows in the category of contexts of $\mathcal{N}$.

Checking types

```
isTy :: Cxt -> Ty -> Bool
isTy cxt (Pi a b) = isTy cxt a && isTy (a:cxt) b
isTy cxt U        = True
isTy cxt a        = isTm cxt U a
```

checks whether an expression is a correct type with respect to a context. Such types will be the "types" of $\mathcal{N}$.

Checking terms

```
isTm :: Cxt -> Ty -> Exp -> Bool
isTm cxt (Pi a b) (Lam t)   = isTm (a:cxt) b t
isTm cxt a        (Lam t)   = False
isTm cxt U        (Pi a b)  = isTm cxt U a && isTm (a:cxt) U b
isTm cxt a        (Pi a b)  = False
isTM cxt a        U         = False
isTm cxt a        s         = case inferTy cxt s of
                                  Just a' -> a == a'
                                  Nothing -> False
```

checks whether an expression has a type with respect to a context. Such terms will be the "terms" of $\mathcal{N}$.

Infering the type of a neutral term. The type-checking algorithm is as usual bi-directional: to check whether an application has a given type we try to infer the type of the function and then check whether it matches the type of the argument.

```
inferTy :: Cxt -> Exp -> Maybe Ty
inferTy cxt (Var i)   = Just (shift (cxt !! i) (i+1))
inferTy cxt (App s t) = case inferTy cxt s of
   Just (Pi a b) -> if isTm cxt a t
                    then Just (subst b (t : ide))
                    else Nothing
   otherwise     -> Nothing
```

This function expects a neutral expression as input and tries to infer its type. It calls an auxiliary function

```
shift :: Exp -> Int -> Exp
shift t i = subst t (map Var [i ..])
```

so that `shift e n` increases all free variables in `e` by `n`.

Implementing the operations of ΠU-cwfs. To perform type inference we also call the "hereditary" substitution function `subst`. This is one of the cwf-combinators. We will now implement them in the order they appear in the above definition of cwf. Note that many of the equations for the cwf-combinators reappear in the programs below, a fact which will facilitate the checking that $\mathcal{N}$ is a cwf.

The empty context is just the empty list and context extension is implemented by the `Cons`-operation on lists. The composition $\circ$ and the identity id combinators are implemented by

```
comp :: Subst -> Subst -> Subst
comp []     ts' = []
comp (t:ts) ts' = (subst t ts'):(comp ts ts')

ide :: Subst
ide = map Var [0 .. ]
```

The length of the identity substitution id_Γ depends on the context Γ, but here we use a lazy infinite list for simplicity. Note that when we check that `ide` is a correct substitution with respect to a context of length n we only check the n first elements of the list `ide`.

Substitution $-[-]$ in types and terms is the same function:

```
subst :: Exp -> Subst -> Exp
subst (Var i)   ts = ts !! i
subst (App s t) ts = app (subst s ts) (subst t ts)
subst (Lam t)   ts = Lam (subst t (lift ts))
subst (Pi a b)  ts = Pi (subst a ts) (subst b (lift ts))
subst U         ts = U
```

where we use the lifting function

```
lift :: Subst -> Subst
lift ts = q : comp ts p
```

which is just an abbreviation of a cwf combinator expression.

The terminal arrow is just the empty list, and substitution extension is just the `Cons` operation on lists. The projections p and q are

```
p :: Subst
p = map Var [1 .. ]

q :: Exp
q = Var 0
```

Like in the case of the identity the length of the substitution $\mathrm{p}_{\Gamma,A}$ depends on the context Γ and for simplicity we implement it by an infinite list.

The type constructor Π and the term constructor λ are implemented by the constructors `Pi` and `Lam`. Application is

```
app :: Exp -> Exp -> Exp
app (Lam t) s = subst t (s:ide)
app r s       = App r s
```

The type constructor U and the term constructor $\hat{\Pi}$ are implemented by the constructors `U` and `Pi`. The decoding function T is implemented by the identity function on expressions. We have a universe à la Russell.

We can now formulate the correctness of our type-checking algorithm as follows: $\mathcal{N}$ is an initial object in $\mathbf{Cwf}_{\mathbf{\Pi U}}$. This states in particular that $\mathcal{N}$ is categorically equivalent to any other initial ΠU-cwf, such as the variable-free substitution calculus obtained by using Cartmell's method for constructing initial objects from generalized algebraic theories, or any traditional presentation of Martin-Löf type theory which we can organize as an ΠU-cwf and prove initial in $\mathbf{Cwf}_{\mathbf{\Pi U}}$. See Hofmann [8] for a description of the correspondence between cwfs and lambda calculus presentations of type theory.

References

1. Abel, A., Aehlig, K., Dybjer, P.: Normalization by evaluation for Martin-Löf type theory with one universe. Electr. Notes Theor. Comput. Sci. 173, 17–39 (2007)
2. Abel, A., Coquand, T., Dybjer, P.: Normalization by evaluation for Martin-Löf type theory with typed equality judgements. In: LICS, pp. 3–12 (2007)
3. Bertot, Y., Castéran, P.: Interactive Theorem Proving and Program Development Coq'Art: The Calculus of Inductive Constructions. Texts in Theoretical Computer Science. An EATCS Series (2004)
4. Cartmell, J.: Generalised algebraic theories and contextual categories. Annals of Pure and Applied Logic 32, 209–243 (1986)
5. Chapman, J., Altenkirch, T., McBride, C.: Epigram reloaded: A standalone typechecker for ETT. In: Proceedings of TFP (July, 2005)
6. Coquand, T.: An algorithm for type-checking dependent types. Sci. Comput. Program. 26(1-3), 167–177 (1996)
7. Dybjer, P.: Internal type theory. In: Berardi, S., Coppo, M. (eds.) TYPES 1995. LNCS, vol. 1158, pp. 120–134. Springer, Heidelberg (1996)
8. Hofmann, M.: Syntax and semantics of dependent types. In: Pitts, A., Dybjer, P. (eds.) Semantics and Logics of Computation, Cambridge University Press, Cambridge (1996)
9. Martin-Löf, P.: Constructive mathematics and computer programming. In: Logic, Methodology and Philosophy of Science, 1979, vol. VI, pp. 153–175. North-Holland, Amsterdam (1982)
10. Martin-Löf, P.: Amendment to intuitionistic type theory. Notes from a lecture given in Göteborg (March, 1986)

11. Martin-Löf, P.: Substitution calculus. Unpublished notes from a lecture in Göteborg (November, 1992)
12. Nordström, B., Petersson, K., Smith, J.: Programming in Martin-Löf's Type Theory: An Introduction. Oxford University Press, Oxford (1990)
13. Norell, U.: Towards a practical programming language based on dependent type theory. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden (September, 2007)