Intuitionistic Model Constructions and Normalization Proofs

Thierry Coquand and Peter Dybjer

February 16, 1996

Abstract

The traditional notions of *strong* and *weak normalization* refer to properties of a binary *reduction relation*. In this paper we explore an alternative approach to normalization, where we bypass the reduction relation and instead focus on the *normalization function*, that is, the function which maps a term into its normal form. We work in an intuitionistic metalanguage, and characterize a normalization function as an algorithm which picks a canonical representative from the equivalence class of convertible terms. Hence we also get a decision algorithm for convertibility.

Such a normalization function can be constructed by building an appropriate model and a function "quote" which inverts the interpretation function. The normalization function is then obtained by composing the quote function with the interpretation function. We also discuss how to get a simple proof of the property that constructors are one-to-one, which usually is obtained as a corollary of Church-Rosser and normalization in the traditional sense.

We illustrate this approach by showing how a glueing model (closely related to the glueing construction used in category theory) gives rise to a normalization algorithm for a combinatory formulation of Gödel System T. We then show how the method extends in a straightforward way when we add cartesian products and disjoint unions (full intuitionistic propositional logic under a Curry-Howard interpretation) and transfinite inductive types such as the Brouwer ordinals.

1 Introduction

There is a striking analogy between computing a program and assigning semantics to it. This analogy is for instance reflected in the similarity between the equations defining the denotational semantics of a language and the rules of evaluation in an environment machine [16, 28, 21].

In this paper we use this analogy to give a semantical treatment of normalization by building a non-standard model, and a function **quote** which maps a semantic object to a normal term representing it. The normalization function **nf** is then obtained by composing **quote** with the interpretation function $[\![\ ]\!]$, which maps a term into its non-standard meaning. This approach to normalization bypasses the binary reduction relation and its traditional properties of strong and weak normalization and Church-Rosser. Instead of these properties we shall use that a term is convertible to the term returned by the normalization function

$$a \ \mathtt{conv} \ \mathbf{nf}\ a$$

and that the normalization function maps convertible terms to equal terms

$$a \ \mathtt{conv} \ a' \rightarrow \mathbf{nf}\ a = \mathbf{nf}\ a'$$

It follows that the normalization function picks a representative (a *normal form*) from each equivalence class of convertible terms

$$a \ \mathtt{conv} \ a' \leftrightarrow \mathbf{nf}\ a = \mathbf{nf}\ a'$$

and hence can be used to decide convertibility by comparing normal forms. Note that this notion of normal form does not refer to a reduction relation and is not necessarily a normal form in the traditional sense.

Our starting point was the reading of two early papers by Martin-Löf, where he introduced this approach in the context of a general discussion of intuitionistic abstractions on the meta level and the notion of definitional equality [18]. He also proved normalization for intuitionistic type theory [19] in this way. While analyzing these ideas, we realized that there was a close connection to the work by Berger and Schwichtenberg [5]. They showed how to get a normalization algorithm (returning long normal forms) for the simply typed $\lambda\beta\eta$-calculus by inverting an interpretation function.

Here we develop this approach for a small functional programming language based on typed combinatory logic. First we study a combinatory version of Gödel System T in section 2. We give its syntax and standard semantics. Then we show how to derive a normalization algorithm by enriching the standard semantics of function types with a syntactic component, which keeps track of normal forms. We then show how to prove that this algorithm yields a decision algorithm for equality by constructing a glueing model, similar to the model used by Lafont [14] in his work on deriving the categorical abstract machine from a proof of termination for a categorical combinator language. Furthermore we show how metamathematical properties, such as consistency and that constructors are one-to-one can be derived. Finally, we show how our glueing construction can be modified to give a proof of weak normalization in the traditional sense and how Church-Rosser then follows as a corollary.

In section 3 we show how our method extends in a straightforward way when finite disjoint unions and cartesian products are added to the language. By the Curry-Howard identification of proposition and types we thus get a model-based proof of normalization for full intuitionistic propositional logic.

In section 4 we show how to extend our method to transfinite inductive types by giving the example of Brouwer ordinals.

It was essential to us (and to Martin-Löf [18]) to think in terms of an intuitionistic metalanguage when developing these ideas. We also wished to show in detail that Martin-Löf's intuitionistic type theory was adequate as a formal metalanguage for this task and implemented our constructions using ALF – an implementation of Martin-Löf type theory [1]. In section 5 we discuss issues which relate specifically to the metalanguage and its implementation. Following suggestions from the referees we have rewritten section 2-4, which were originally derived from a formal ALF-development, in a more informal style to make them accessible to readers unfamiliar with Martin-Löf type theory.

In section 6 we discuss related work.

2 Gödel System T

2.1 Syntax

We begin by defining the set $\mathtt{Type}$ of types inductively. This set is built up by the following two rules

- if $A, B \in \mathtt{Type}$ then $A \Rightarrow B \in \mathtt{Type}$,
- $\mathtt{N} \in \mathtt{Type}$.

We use infix right associative notation for $\Rightarrow$.

We then define the sets of terms $\mathtt{T}(A)$ indexed by $A \in \mathtt{Type}$. It is inductively generated by the following rules

- $\mathtt{K}(A, B) \in \mathtt{T}(A \Rightarrow B \Rightarrow A)$ if $A, B \in \mathtt{Type}$,
- $\mathtt{S}(A, B, C) \in \mathtt{T}((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C)$ if $A, B \in \mathtt{Type}$,
- $\mathtt{app}(A, B, c, a) \in \mathtt{T}(B)$ if $A, B \in \mathtt{Type}$, $c \in \mathtt{T}(A \Rightarrow B)$ and $a \in \mathtt{T}(A)$,
- $\mathtt{0} \in \mathtt{T}(\mathtt{N})$,
- $\mathtt{s}(a) \in \mathtt{T}(\mathtt{N})$ if $a \in \mathtt{T}(\mathtt{N})$,
- $\mathtt{rec}(C, d, e) \in \mathtt{T}(\mathtt{N} \Rightarrow C)$ if $C \in \mathtt{Type}$, $d \in \mathtt{T}(C)$ and $e \in \mathtt{T}(\mathtt{N} \Rightarrow C \Rightarrow C)$.

We shall use polymorphic notation whenever appropriate for improving readability. For example, we write $\mathtt{K}$, $\mathtt{S}$, $\mathtt{app}(c,a)$, $\mathtt{rec}(d,e)$ for $\mathtt{K}(A,B)$, $\mathtt{S}(A,B,C)$, $\mathtt{app}(A,B,c,a)$, and $\mathtt{rec}(C,d,e)$ respectively.

Note that we directly generate the *well-typed* terms of each type. This is different from the traditional approach, where one first introduces a set of *raw terms* and then defines a binary typing relation between raw terms and types. We discuss this point further in section 6.1.

2.2 Intended semantics

In the intended semantics we interpret a type A as a set $[\![A]\!]$:

$$\begin{aligned} [\![A \Rightarrow B]\!] &= [\![A]\!] \to [\![B]\!] \\ [\![\mathtt{N}]\!] &= N \end{aligned}$$

The interpretation $[\![a]\!]_A \in [\![A]\!]$ of an object $a \in \mathtt{T}(A)$ is defined by recursion on the inductive generation of a:

$$\begin{aligned} [\![\mathtt{K}]\!] &= \lambda x.\lambda y.x \\ [\![\mathtt{S}]\!] &= \lambda g.\lambda f.\lambda x.g\ x\ (f\ x) \\ [\![\mathtt{app}(c,a)]\!] &= [\![c]\!]\ [\![a]\!] \\ [\![\mathtt{0}]\!] &= 0 \\ [\![\mathtt{s}(a)]\!] &= s\ [\![a]\!] \\ [\![\mathtt{rec}(d,e)]\!] &= rec\ [\![d]\!]\ [\![e]\!] \end{aligned}$$

We work in the intuitionistic metalanguage of Martin-Löf type theory, and our intended semantics can thus be viewed as an intuitionistic version of the standard (Tarski) set-theoretic semantics of Gödel System T. However, the reader who prefers to look at metalanguage expressions as in informal mathematics or as in formal set theory should have no problem, since Martin-Löf type theory has a straightforward "naive" set-theoretic semantic [11].

We point out that there is a close parallel between object language expressions (of Gödel System T) and metalanguage notations used for their interpretation. We use the convention that object language expressions are written in `teletype` and metalanguage expressions in *italic*. Moreover, we have arbitrarily chosen to use logical symbols in the object language ($\Rightarrow, \bot, \top, \vee, \wedge$) and type forming symbols in the metalanguage ($\to, \emptyset, 1, +, \times$).

For example, the set N of natural numbers in the metalanguage are generated inductively by 0 and s. We have a metalanguage primitive recursion operator

$$rec_C \in C \to (N \to C \to C) \to N \to C$$

defined by

$$\begin{aligned} rec\ d\ e\ 0 &= d \\ rec\ d\ e\ (s\ a) &= e\ a\ (rec\ d\ e\ a) \end{aligned}$$

Moreover, we introduce conversion as a family of relations indexed by the types: $a\ \mathtt{conv}_A\ a'$ for $A \in \mathtt{Type}$ and $a, a' \in \mathtt{T}(A)$. (Also here we often omit the index and write $a\ \mathtt{conv}\ a'$.) This relation is inductively generated by the following rules:

$$a\ \mathtt{conv}\ a$$

$$\frac{a\ \mathtt{conv}\ a' \qquad a'\ \mathtt{conv}\ a''}{a\ \mathtt{conv}\ a''}$$

$$\frac{a\ \mathtt{conv}\ b}{b\ \mathtt{conv}\ a}$$

$$\frac{c \ \mathtt{conv}\ c' \qquad a \ \mathtt{conv}\ a'}{\mathtt{app}(c, a) \ \mathtt{conv}\ \mathtt{app}(c', a')}$$

$$\frac{a \ \mathtt{conv}\ a'}{\mathtt{s}(a) \ \mathtt{conv}\ \mathtt{s}(a')}$$

$$\frac{d \ \mathtt{conv}\ d' \qquad e \ \mathtt{conv}\ e'}{\mathtt{rec}(d, e) \ \mathtt{conv}\ \mathtt{rec}(d', e')}$$

$$\mathtt{app}(\mathtt{app}(K, a), b) \ \mathtt{conv}\ a$$

$$\mathtt{app}(\mathtt{app}(\mathtt{app}(\mathtt{S}, g), f), a) \ \mathtt{conv}\ \mathtt{app}(\mathtt{app}(g, a), \mathtt{app}(f, a))$$

$$\mathtt{app}(\mathtt{rec}(d, e), \mathtt{0}) \ \mathtt{conv}\ d$$

$$\mathtt{app}(\mathtt{rec}(d, e), \mathtt{s}(a)) \ \mathtt{conv}\ \mathtt{app}(\mathtt{app}(e, a), \mathtt{app}(\mathtt{rec}(d, e), a))$$

Theorem 1 *If* $a \ \mathtt{conv}\ a'$ *then* $[\![a]\!] = [\![a']\!]$.

Proof. By induction on the proof that $a \ \mathtt{conv}\ a'$. □

Corollary 2 *Gödel System T is equationally consistent, that is, it has no derivation of* $\mathtt{0} \ \mathtt{conv}\ \mathtt{s}(\mathtt{0})$.

Proof. If we assume $\mathtt{0} \ \mathtt{conv}\ \mathtt{s}(\mathtt{0})$, then we can prove $0 = 1$ – a contradiction. □

However, to prove for example that the constructor $\mathtt{s}$ is one-to-one for convertibility we need the normalization proof.

2.3 Normalization algorithm

The interpretation function into the intended model is not injective and hence cannot be inverted. (The intended model is a λ-algebra and all combinatory algebras are not λ-algebras. So there are terms which are identified in the model but not by convertibility, see Barendregt [3]).

However, if we enrich the interpretation of $\Rightarrow$ to have a syntactic as well as a semantic component:

$$\begin{aligned} [\![A \Rightarrow B]\!] &= \mathtt{T}(A \Rightarrow B) \times ([\![A]\!] \to [\![B]\!]) \\ [\![\mathtt{N}]\!] &= N \end{aligned}$$

then the function

$$\mathbf{quote}_A \ \in [\![A]\!] \to \mathtt{T}(A)$$

defined by

$$\begin{aligned} \mathbf{quote}_{A \Rightarrow B} \ \langle c, f \rangle &= c \\ \mathbf{quote}_{\mathtt{N}} \ 0 &= \mathtt{0} \\ \mathbf{quote}_{\mathtt{N}} \ (s \ p) &= \mathtt{s}(\mathbf{quote}_{\mathtt{N}} \ p) \end{aligned}$$

inverts the interpretation function

$$[\![\ ]\!]_A \in \mathtt{T}(A) \to [\![A]\!]$$

defined by

$$\begin{array}{rcl}
[\![\mathtt{K}]\!] & = & \langle \mathtt{K}, \lambda p.\langle \mathtt{app}(\mathtt{K}, \mathbf{quote}\ p), \lambda q.p\rangle\rangle \\
[\![\mathtt{S}]\!] & = & \langle \mathtt{S}, \lambda p.\langle \mathtt{app}(\mathtt{S}, \mathbf{quote}\ p), \lambda q.\langle \mathtt{app}(\mathtt{app}(\mathtt{S}, \mathbf{quote}\ p), \mathbf{quote}\ q), \lambda r.app_M\ (app_M\ p\ r)(app_M\ q\ r)\rangle\rangle\rangle \\
[\![\mathtt{app}(c,a)]\!] & = & app_M\ [\![c]\!]\ [\![a]\!] \\
[\![\mathtt{0}]\!] & = & 0 \\
[\![\mathtt{s}(a)]\!] & = & s\ [\![a]\!] \\
[\![\mathtt{rec}(d,e)]\!] & = & \langle \mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\rangle
\end{array}$$

where we have used the following application operator in the model:

$$app_M\ \langle c, f\rangle\ q \quad = \quad f\ q$$

The normal form function can then be defined by

$$\mathbf{nf}\ a \quad = \quad \mathbf{quote}\ [\![a]\!]$$

Theorem 3 *If* a `conv` a' *then* $\mathbf{nf}\ a\ =\ \mathbf{nf}\ a'$.

Proof. We can check the equalities

$$\begin{array}{rcl}
[\![\mathtt{app}(\mathtt{app}(\mathtt{K}, a), b)]\!] & = & [\![a]\!] \\
[\![\mathtt{app}(\mathtt{app}(\mathtt{app}(\mathtt{S}, g), f), a)]\!] & = & [\![\mathtt{app}(\mathtt{app}(g, a), \mathtt{app}(f, a))]\!] \\
[\![\mathtt{app}(\mathtt{rec}(d, e), \mathtt{0})]\!] & = & d \\
[\![\mathtt{app}(\mathtt{rec}(d, e), \mathtt{s}(a))]\!] & = & [\![\mathtt{app}(\mathtt{app}(e, a), \mathtt{app}(\mathtt{rec}(d, e), a))]\!]
\end{array}$$

by pure equational reasoning. It hence follows that $[\![a]\!] = [\![a']\!]$ whenever a `conv` a', by induction on the proof that a `conv` a', and hence that $\mathbf{nf}\ a\ =\ \mathbf{nf}\ a'$ whenever a `conv` a'. □

2.4 The normalization algorithm in ML

As pointed out to us by Thorsten Altenkirch, this program can be translated into SML [22], and provides then a concise and elegant implementation of combinatory reduction.

```
datatype tm = s | k | ap of tm*tm;
datatype vl = v_arr of tm*(vl->vl);

fun term_part (v_arr (M,_)) = M;
fun val_ap (v_arr (_,f),x) = f x;

fun eval s =
  v_arr (s,fn x => v_arr (ap(s,term_part x),
                          fn y => v_arr (ap (ap (s,term_part x),term_part y),
                                         fn z => val_ap (val_ap (x,z),
                                                         val_ap (y,z)))))
  | eval k = v_arr (k,fn x => v_arr (ap(k,term_part x),
                                     fn y => x))
  | eval (ap(x,y)) = val_ap (eval x,eval y);

fun norm t = term_part (eval t);
```

2.5 Normalization proof

Theorem 4 a `conv` $\mathbf{nf}\ a$, *that is,* $\mathbf{quote}$ *is a section of* $[\![\ ]\!]$.

Proof. We use *initial algebra semantics* to structure our proof. A model of Gödel System T is a typed combinatory algebra extended with operations for interpreting $\mathtt{0}$, $\mathtt{s}$, and $\mathtt{rec}$, such that the two equations for primitive recursion are satisfied. The syntactic algebra of terms $\mathtt{T}(A)$ under convertibility $\mathtt{conv}_A$ is an initial model. The non-standard model described in 2.3 is another model. The interpretation function $[\![\ ]\!]_A : \mathtt{T}(A) \to [\![A]\!]$ is the unique homomorphism from the initial model into the non-standard model. If we could prove that $\mathbf{quote}_A : [\![A]\!] \to \mathtt{T}(A)$ also is a homomorphism, then it would follow that $\mathbf{nf}_A : \mathtt{T}(A) \to \mathtt{T}(A)$, which is the composition of $[\![\ ]\!]_A$ and $\mathbf{quote}_A$, is also a homomorphism. Hence it must be equal to the identity homomorphism, and hence $a\ \mathtt{conv}\ \mathbf{nf}\ a$.

But **quote** does not preserve application. However, we can construct a submodel of the non-standard model, such that the restriction of $\mathbf{quote}_A$ to this submodel is a homomorphism. We call this the *glued* submodel, since it is closely related to the glueing construction in category theory, see section 2.6 and Lafont [14].

In the submodel we require that a value $p \in [\![A]\!]$ satisfies the property $Gl_A\ p$ defined by induction on the type A:

- $Gl_{A \Rightarrow B}\ q$ holds *iff* for all $p \in [\![A]\!]$ if $Gl_A\ p$ then
  1. $Gl_B(app_M\ q\ p)$ and
  2. $\mathtt{app}(\mathbf{quote}\ q, \mathbf{quote}\ p)\ \mathtt{conv}\ (\mathbf{quote}\ app_M\ q\ p)$
- $Gl_N\ n$ for all $n \in N$.

Lemma 5 *For all $a \in \mathtt{T}(A)$ $Gl_A\ [\![a]\!]$. Hence, the submodel of glued values indeed is a model of Gödel System T.*

Proof: By induction on a. We show the cases for $\mathtt{K}$ and $\mathtt{rec}$. The other cases are similar or trivial.
Case $\mathtt{K}$. We wish to prove

$$Gl_{A \Rightarrow B \Rightarrow A}\ \langle \mathtt{K}, \lambda p.\langle \mathtt{app}(\mathtt{K}, \mathbf{quote}\ p), \lambda q.p\rangle\rangle$$

But this is immediate by unfolding the definition of $Gl_{A \Rightarrow B \Rightarrow A}$ and using

$$\mathtt{app}(\mathtt{app}(\mathtt{K}, \mathbf{quote}\ p), \mathbf{quote}\ q)\ \mathtt{conv}\ \mathbf{quote}\ p$$

Case $\mathtt{rec}$. We wish to prove

$$Gl_{\mathtt{N} \Rightarrow C}\ \langle \mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\rangle$$

from the induction hypotheses $Gl_C\ [\![d]\!]$ and $Gl_{\mathtt{N} \Rightarrow C \Rightarrow C}\ [\![e]\!]$.

So let $Gl_N\ n$ and prove by induction on n that

1. $Gl_C\ (rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\ n)$ and
2. $\mathtt{app}(\mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), \mathbf{quote}\ n)\ \mathtt{conv}\ (\mathbf{quote}\ (rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\ n))$

The base case $n = 0$ is clear:

1. follows from the induction hypothesis, and
2. $\mathtt{app}(\mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), \mathtt{0})\ \mathtt{conv}\ (\mathbf{quote}\ [\![d]\!])$

The induction step $n = s\ m$ follows because

1. follows from the induction hypotheses, and
2.

$$\begin{array}{ll} & \mathtt{app}(\mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), \mathbf{quote}\ (s\ m)) \\ \mathtt{conv} & \mathtt{app}(\mathtt{app}(\mathbf{quote}\ [\![e]\!], \mathbf{quote}\ m), \mathtt{app}(\mathtt{rec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), \mathbf{quote}\ m)) \\ \mathtt{conv} & \mathtt{app}(\mathtt{app}(\mathbf{quote}\ [\![e]\!], \mathbf{quote}\ m), \mathbf{quote}\ (rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\ n)) \\ \mathtt{conv} & \mathbf{quote}\ (app_M\ (app_M\ [\![e]\!]\ m)\ (rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\ m)) \\ \mathtt{conv} & \mathbf{quote}\ (rec\ [\![d]\!]\ (\lambda x.\lambda y.app_M\ (app_M\ [\![e]\!]\ x)\ y)\ (s\ m)) \end{array}$$

□

Lemma 6 **quote** *is a homomorphism from the algebra of glued values to the term algebra.*

The definition of glued value is such that **quote** commutes with `app`. The other cases are also trivial. It now follows from lemma 5 and 6 that **nf** is an identity homomorphism as explained above. □

Corollary 7 *a* `conv` *b iff* **nf** *a* = **nf** *b.*

Proof. **nf** *a* = **nf** *b* implies *a* `conv` *b* follows from theorem 4. The reverse implication is theorem 3. □

2.6 Categorical glueing

The glueing construction in category theory builds a new category [15, 23] from a functor $T : \mathcal{C} \to \mathcal{S}$ with objects arrows

$$X \xrightarrow{q_A} TA$$

and arrows pairs $\langle c, f \rangle$, such that the following square in $\mathcal{S}$ commutes:

$$\begin{array}{ccc} X & \xrightarrow{q_A} & TA \\ {\scriptstyle f}\downarrow & & \downarrow{\scriptstyle Tc} \\ Y & \xrightarrow[q_B]{} & TB \end{array}$$

The Freyd cover is the special case of where $\mathcal{S}$ is the category of sets and $TA = \mathcal{C}(1, A)$.

We have chosen notation to suggest the connection between our model of glued values for combinatory logic and the glueing construction: the commuting square in the definition of arrow in the glued category is reminiscent of the the definition of glued value for a function type.

2.7 Weak normalization revisited

We end this section by reintroducing the relation `red` of reduction (in zero or more steps) and point out that the normalization proof in our sense can easily be modified to a proof of weak normalization in the traditional sense. We modify our glueing model so that instead

- $Gl_{A \Rightarrow B}\ q$ holds *iff* for all $p \in [\![A]\!]$ if $Gl_A\ p$ then
  1. $Gl_B(app_M\ q\ p)$ and
  2. `app`(**quote** q, **quote** p) `red` (**quote** $app_M\ q\ p$)
- $Gl_N\ n$ for all $n \in N$.

Theorem 8 *Weak normalization: a* `red` **nf** *a and* **nf** *a is irreducible.*

Proof. The proof of *a* `conv` **nf** *a* is easily modified to a proof that *a* `red` **nf** *a*.

It remains to prove that **nf** *a* is irreducible. But if let $\mathbf{Nf}(A) \subseteq \mathtt{T}(A)$ be the set of irreducible terms, and we redefine the interpretation so that

$$[\![A \Rightarrow B]\!] \quad = \quad \mathbf{Nf}(A \Rightarrow B) \times ([\![A]\!] \to [\![B]\!])$$

then we can verify that **quote** and **nf** have the following types

$$\begin{array}{rcl} \mathbf{quote}_A & \in & [\![A]\!] \to \mathbf{Nf}(A) \\ \mathbf{nf}_A & \in & \mathtt{T}(A) \to \mathbf{Nf}(A) \end{array}$$

□

Corollary 9 *Church-Rosser: if a `red` b and a `red` b' then there is c such that b `red` c and b' `red` c.*

Proof (due to Peter Hancock, see Martin-Löf [19]). It follows that b `conv` b' and hence by theorem 3 that $\mathbf{nf}\ b = \mathbf{nf}\ b'$. Hence let $c = \mathbf{nf}\ b = \mathbf{nf}\ b'$ and by theorem 8 b `red` c and b' `red` c. □

3 Propositional calculus

3.1 Syntax

We now extend our object language with finite disjoint unions and finite cartesian products, so that we get full intuitionistic propositional calculus by the Curry-Howard identification of propositions and types.

The new inductive clauses for `Type` are:

- $\bot \in$ `Type`,
- $\top \in$ `Type`,
- $A \wedge B \in$ `Type` if $A, B \in$ `Type`,
- $A \vee B \in$ `Type` if $A, B \in$ `Type`.

There are also new term constructors corresponding to the introduction and elimination rules for propositional calculus

- `case0`$(C) \in$ `T`$(\bot \Rightarrow C)$ if $C \in$ `Type`,
- `<>` $\in$ `T`$(\top)$,
- `inl`$(A, B, a) \in$ `T`$(A \vee B)$ if $A, B \in$ `Type` and $a \in$ `T`(A),
- `inr`$(A, B, b) \in$ `T`$(A \vee B)$ if $A, B \in$ `Type` and $b \in$ `T`(B),
- `case`$(A, B, C, d, e) \in$ `T`$(A \vee B \Rightarrow C)$ if $A, B, C \in$ `Type` and $d \in$ `T`$(A \Rightarrow C),\ e \in$ `T`$(B \Rightarrow C))$,
- `<`a, b`>` $\in$ `T`$(A \wedge B)$ if $A, B \in$ `Type` and $a \in$ `T`$(A), b \in$ `T`(B),
- `fst`$(A, B, c) \in$ `T`(A) if $A, B \in$ `Type` and $c \in$ `T`$(A \wedge B)$,
- `snd`$(A, B, c) \in$ `T`(B) if $A, B \in$ `Type` and $c \in$ `T`$(A \wedge B)$.

As before, we adopt polymorphic notation for these constants; for instance `inl`(a) abbreviates `inl`(A, B, a) if $A, B \in$ `Type` and $a \in$ `T`(A).

3.2 Intended semantics

We have the following intended meaning of the new types

$$\begin{aligned} [\![\bot]\!] &= \emptyset \\ [\![\top]\!] &= 1 \\ [\![A \vee B]\!] &= [\![A]\!] + [\![B]\!] \\ [\![A \wedge B]\!] &= [\![A]\!] \times [\![B]\!] \end{aligned}$$

and of the new terms

$$\begin{aligned} [\![\texttt{case0}]\!] &= \mathit{case}_0 \\ [\![\texttt{<>}]\!] &= \langle\rangle \\ [\![\texttt{inl}(a)]\!] &= \mathit{inl}\ [\![a]\!] \\ [\![\texttt{inr}(b)]\!] &= \mathit{inr}\ [\![b]\!] \end{aligned}$$

$$
\begin{array}{rcl}
[\![\texttt{case}(d,e)]\!] & = & case\ [\![d]\!]\ [\![e]\!] \\
[\![\texttt{<}a,b\texttt{>}]\!] & = & \langle [\![a]\!], [\![b]\!] \rangle \\
[\![\texttt{fst}(c)]\!] & = & fst\ [\![c]\!] \\
[\![\texttt{snd}(c)]\!] & = & snd\ [\![c]\!]
\end{array}
$$

We hope that metalanguage notations are clear. For example,

$$case_{A,B,C} \in (A \to C) \to (B \to C) \to (A + B) \to C$$

is the function defined by

$$
\begin{array}{rcl}
case\ d\ e\ (inl\ a) & = & d\ a \\
case\ d\ e\ (inr\ a) & = & e\ b
\end{array}
$$

Theorem 10 *Logical consistency: there is no element (proof) in* $\mathtt{T}(\bot)$.

Proof. If we specialize the term interpretation function to $\bot$ we get a function in $\mathtt{T}(\bot) \to \emptyset$. Hence assuming that there is an element in $\mathtt{T}(\bot)$ leads to a contradiction. □

This proof can be seen as an internalization of the proof of "simple-minded" consistency in Martin-Löf [20].

We next extend the definition of conversion with the following rules

$$
\begin{array}{rcl}
\texttt{app}(\texttt{case}(d,e),\texttt{inl}(a)) & \texttt{conv} & \texttt{app}(d,a) \\
\texttt{app}(\texttt{case}(d,e),\texttt{inr}(b)) & \texttt{conv} & \texttt{app}(e,b) \\
\texttt{fst}(\texttt{<}a,b\texttt{>}) & \texttt{conv} & a \\
\texttt{snd}(\texttt{<}a,b\texttt{>}) & \texttt{conv} & b
\end{array}
$$

together with congruence rules for the new term constructors. We can check that these rules are valid under the intended semantics.

3.3 Normalization algorithm

We can extend the non-standard model in 2.3, which can be used for normalization, by interpreting the new type formers as in the intended semantics.

We also extend the definition of quote:

$$
\begin{array}{rcl}
\mathbf{quote}_{\top}\ \langle\rangle & = & \texttt{<>} \\
\mathbf{quote}_{A \vee B}\ (inl\ p) & = & \texttt{inl}(\mathbf{quote}_A\ p) \\
\mathbf{quote}_{A \vee B}\ (inr\ q) & = & \texttt{inr}(\mathbf{quote}_B\ q) \\
\mathbf{quote}_{A \wedge B}\ \langle p, q\rangle & = & \texttt{<}\mathbf{quote}_A\ p, \mathbf{quote}_B\ q\texttt{>}
\end{array}
$$

We also need to extend the interpretation function for terms in the enriched model (we omit cases which are the same as for the intended interpretation).

$$
\begin{array}{rcl}
[\![\texttt{case0}]\!] & = & \langle \texttt{case0}, case_0 \rangle \\
[\![\texttt{case}(d,e)]\!] & = & \langle \texttt{case}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), case\ (\lambda x.app_M\ [\![d]\!]\ x)\ (\lambda y.app_M\ [\![e]\!]\ y) \rangle
\end{array}
$$

3.4 Normalization proof

We extend the definition of Gl as follows.

- $Gl_{\top}\ \langle\rangle$ is true.
- $Gl_{A \vee B}\ (inl\ p)$ iff $Gl_A\ p$; $Gl_{A \vee B}\ (inr\ q)$ iff $Gl_B\ q$.
- $Gl_{A \wedge B}\ \langle p, q\rangle$ iff $Gl_A\ p$ and $Gl_B\ q$.

The normalization proof in 2.5 can then be extended without difficulty.

4 Brouwer ordinals

4.1 Syntax

We finally show that the methods extends to tranfinite inductive types by giving the example of Brouwer ordinals:

$$\mathtt{Ord} \;\in\; \mathtt{Type}.$$

The new constructors for terms are

- $\mathtt{0} \in \mathtt{T}(\mathtt{Ord})$,
- $\mathtt{sup}(b) \in \mathtt{T}(\mathtt{Ord})$ if $b \in \mathtt{T}(\mathtt{N} \Rightarrow \mathtt{Ord})$,
- $\mathtt{ordrec}(C, d, e) \in \mathtt{T}(\mathtt{Ord} \Rightarrow C)$ if $d \in \mathtt{T}(C)$ and $e \in \mathtt{T}((\mathtt{N} \Rightarrow \mathtt{Ord}) \Rightarrow (\mathtt{N} \Rightarrow C) \Rightarrow C)$.

4.2 Intended semantics

$$\begin{array}{rcl} [\![\mathtt{Ord}]\!] & = & \mathcal{O} \\ [\![\mathtt{0}]\!] & = & 0 \\ [\![\mathtt{sup}(b)]\!] & = & \mathit{sup}\ [\![b]\!] \\ [\![\mathtt{ordrec}(d, e)]\!] & = & \mathit{ordrec}\ [\![d]\!]\ [\![e]\!] \end{array}$$

Here the metalanguage Brouwer ordinals $\mathcal{O}$ is the set inductively generated by the rules

- $0 \in \mathcal{O}$;
- $\mathit{sup}\ b \in \mathcal{O}$ if $b \in N \to \mathcal{O}$.

We thus have a recursion operator for ordinals:

$$\mathit{ordrec}_C \in C \to ((N \to \mathcal{O}) \to (N \to C) \to C) \to \mathcal{O} \to C$$

defined by

$$\begin{array}{rcl} \mathit{ordrec}\ d\ e\ 0 & = & d \\ \mathit{ordrec}\ d\ e\ (\mathit{sup}\ b) & = & e\ b\ (\lambda x.\mathit{ordrec}\ d\ e\ (b\ x)) \end{array}$$

In the object language we have the new conversion rules

$$\begin{array}{rcl} \mathtt{app}(\mathtt{ordrec}(d, e), \mathtt{0}) & \mathtt{conv} & d \\ \mathtt{app}(\mathtt{ordrec}(d, e), \mathtt{sup}(b)) & \mathtt{conv} & \mathtt{app}(\mathtt{app}(e, b), \mathtt{ordrec}(d, e) \circ b) \end{array}$$

together with congruence rules for the new term constructors. Here we have used an auxiliary *syntactic* binary composition operator $c \circ b \in \mathtt{T}(A \Rightarrow C)$ if $c \in \mathtt{T}(B \Rightarrow C),\ b \in \mathtt{T}(A \Rightarrow B))$ defined by

$$c \circ b = \mathtt{app}(\mathtt{app}(\mathtt{S}, \mathtt{app}(\mathtt{K}, c)), b).$$

4.3 Normalization algorithm

To extend our normalization algorithm, we introduce a new model $\mathcal{O}_M$ of the Brouwer ordinals. This model is obtained from the inductive definition of the semantic Brouwer ordinals $\mathcal{O}$. As for function spaces in the non-standard model, we introduce a syntactic component as a new argument to the sup-constructor yielding the following inductive definition of $\mathcal{O}_M$:

- $0_M \in \mathcal{O}_M$,
- $\mathit{sup}_M\ c\ f \in \mathcal{O}_M$ if $c \in \mathtt{T}(\mathtt{N} \Rightarrow \mathtt{Ord})$ and $f \in N \to \mathcal{O}_M$.

It follows that we have a recursion operator for $\mathcal{O}_M$:

$$ordrec_M \in C \to (\mathtt{T}(\mathtt{N} \Rightarrow \mathtt{Ord}) \to (N \to \mathcal{O}_M) \to (N \to C) \to C) \to \mathcal{O}_M \to C$$

defined by

$$\begin{array}{rcl} ordrec_M\ d\ e\ 0_M & = & d \\ ordrec_M\ d\ e\ (sup_M\ c\ f) & = & e\ c\ f\ (\lambda x.ordrec_M\ d\ e\ (f\ x)) \end{array}$$

Now we are ready to extend $[\![\ ]\!]$ and **quote**:

$$[\![\mathtt{Ord}]\!] \quad = \quad \mathcal{O}_M$$

The quote function has the following new clauses:

$$\begin{array}{rcl} \mathbf{quote}_{\mathtt{Ord}}\ 0_M & = & \mathtt{0} \\ \mathbf{quote}_{\mathtt{Ord}}\ (sup_M\ c\ f) & = & \mathtt{sup}(c) \end{array}$$

The term interpretation function in the enriched model has the following new clauses:

$$\begin{array}{rcl} [\![\mathtt{0}]\!] & = & 0_M, \\ [\![\mathtt{sup}(b)]\!] & = & sup_M\ (\mathbf{quote}\ [\![b]\!])\ (\lambda x.app_M\ [\![b]\!]\ x), \\ [\![\mathtt{ordrec}(d,e)]\!] & = & \langle \mathtt{ordrec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]), \\ & & ordrec_M\ [\![d]\!]\ (\lambda x.\lambda y.\lambda z.app_M\ (app_M\ [\![e]\!]\ \langle x,y\rangle)\ \langle \mathtt{ordrec}(\mathbf{quote}\ [\![d]\!], \mathbf{quote}\ [\![e]\!]) \circ x, z\rangle\rangle \end{array}$$

4.4 Normalization proof

We extend the definition of glued value to ordinals. The property $Gl_{\mathcal{O}}$ of elements of $\mathcal{O}_M$ is defined inductively by the following rules

- $Gl_{\mathcal{O}}\ 0_M$,;
- $Gl_{\mathcal{O}}\ (sup_M\ c\ f)$ *iff* $c \in \mathtt{T}(\mathtt{N} \Rightarrow \mathtt{Ord})$, $f \in N \to \mathcal{O}_M$, and for all $p \in N$,
  1. $Gl_{\mathcal{O}}(f\ p)$ and
  2. $\mathtt{app}(c, \mathbf{quote}\ p)\ \mathtt{conv}\ \mathbf{quote}\ (f\ p))$.

  The normalization proof in 2.5 extends.

5 Constructors are one-to-one

Theorem 11 *If* $\mathtt{s}(a)\ \mathtt{conv}\ \mathtt{s}(b)$ *then* $a\ \mathtt{conv}\ b$.

Proof. Since **nf** is a homomorphism, the constructor s commutes with the normalization function: $\mathbf{nf}\ (\mathtt{s}(a))\ \mathtt{conv}\ \mathtt{s}(\mathbf{nf}\ a)$. Even more, by definition of $\mathbf{quote}_{\mathtt{N}}$, the constructor s commutes with the normalization function up to equality of terms. Indeed,

$$\begin{array}{rcl} \mathbf{nf}\ (\mathtt{s}(a)) & = & \mathbf{quote}\ [\![\mathtt{s}(a)]\!] \\ & = & \mathbf{quote}\ (s\ [\![a]\!]) \\ & = & \mathtt{s}(\mathbf{quote}\ [\![a]\!]) \\ & = & \mathtt{s}(\mathbf{nf}\ a). \end{array}$$

Let us assume that $a, b \in \mathtt{T}(\mathtt{N})$ satisfy $\mathtt{s}(a)\ \mathtt{conv}\ \mathtt{s}(b)$. Hence,

$$\mathbf{nf}\ (\mathtt{s}(a)) = \mathbf{nf}\ (\mathtt{s}(b))$$

since conversion implies identity of normal forms. Hence,

$$\mathbf{s}(\mathbf{nf}\ a) = \mathbf{s}(\mathbf{nf}\ b)$$

by commutation of **nf** and **s**, and finally

$$\mathbf{nf}(a) = \mathbf{nf}(b)$$

(The constructor **s** is one-to-one for the equality of the meta-language!) From this follows

$$a\ \mathtt{conv}\ b$$

because any term is convertible to its normal form. □

Theorem 12 *If* $\mathtt{sup}(b)$ $\mathtt{conv}$ $\mathtt{sup}(b')$ *then* b $\mathtt{conv}$ b'.

Proof. What is crucial is that the constructor **sup** commutes with normalization up to syntactic equality:

$$\begin{aligned}
\mathbf{nf}\ (\mathtt{sup}(b)) &= \mathbf{quote}\ [\![\mathtt{sup}(b)]\!] \\
&= \mathbf{quote}\ (sup_M\ (\mathbf{quote}\ [\![b]\!])\ (\lambda x.app_M\ [\![b]\!]\ x)) \\
&= \mathtt{sup}(\mathbf{quote}\ [\![b]\!]) \\
&= \mathtt{sup}(\mathbf{nf}\ b).
\end{aligned}$$

□

The usual way of proving the fact that constructors are one-to-one for conversion is to rely on the Church-Rosser's property.

6 The role of an intuitionistic metalanguage

Why do we emphasize that we work in an intuitionistic metalanguage? The reason is of course conceptual rather than formal. Indeed, we have presented the technical development in an informal mathematical style readily understandable from a classical as well as an intuitionistic point of view.

The most fundamental fact is that intuitionistically a function is the same as an algorithm, so a normalization function is as good as step-by-step-reduction for representing a mechanical procedure. Furthermore, intuitionistically a proof that each term has a normal form is a function which given a term returns a normal term (and a proof that the returned term is a normal form). This is one way of deriving a normalization function: by extracting the computational content of a normalization proof. This is based on the Brouwer-Heyting-Kolmogorov-interpretation of proofs in intuitionistic logic and of the Curry-Howard isomorphism between propositions and types.

The reader is referred to Martin-Löf [18] for further discussion of the role of the intuitionistic metalanguage. We started our investigations while trying to answer the question: what is an elegant approach to normalization provided Martin-Löf's intuitionistic type theory is used as a metalanguage? We also wished to experiment with computer-assisted proofs using the proof assistant ALF [1] for intuitionstic type theory and hoped that formalization would throw light on for example the following statement [18]:

> The transition to intuitionistic abstractions on the metalevel is both essential and non-trivial. Essential, because in what seems to me to be the most fruitful notion of model, the interpretation of the convertibility relation **conv**, is standard, that is, it is interpreted as definitional equality $=_{def}$ in the model, and definitional equality is a notion which is unmentionable within the classical set theoretic framework.

Below we shall discuss some aspects of the formalization which were specifically guided by the structure of Martin-Löf type theory: the representation of syntax, different notions of equality, and the representation of algebraic notions in type theory.

The reader is referred to the following books and papers on Martin-Löf type theory: Martin-Löf [20] and Nordström, Petersson, and Smith [25] for general information, Dybjer [12] on inductive definitions in type theory, and to Coquand [7] on definition by pattern-matching in type theory.

6.1 Non-standard syntax

Our presentation of the syntax of typed combinatory logic was non-standard in its use of dependent types. In this way, each subterm is decorated with its type. An alternative presentation would have been to define first a set of *raw* terms $\mathtt{Raw}$, with an untyped conversion relation $t\ \mathtt{conv}\ t'$, as done usually in combinatory logic [13], and then to introduce a typing relation $t\ \epsilon\ A$ between raw terms $t \in \mathtt{Raw}$ and types $A \in \mathtt{Type}$. It is direct to define a map $\mathtt{strip}_A \in \mathtt{T}(A) \to \mathtt{Raw}$ which strips a typed term from its type decorations. The following proposition is then proved by induction:

Proposition 13 *If* $a \in \mathtt{T}(A)$, *then* $\mathtt{strip}\ a\ \epsilon\ A$. *Furthermore, if* $a' \in \mathtt{T}(A)$ *and* $a\ \mathtt{conv}_A\ a'$, *then* $\mathtt{strip}\ a\ \mathtt{conv}\ \mathtt{strip}\ a'$.

This alternative presentation with raw terms is actually the one we have used informally, when choosing a polymorphic notation to represent our terms. A natural question is then: if we use a raw expression t to represent an expression of a given type $a \in \mathtt{T}(A)$, are we sure that this expression does represent such a term in an unambiguous way? It is first easy to prove by induction:

Proposition 14 *If* $t \in \mathtt{Raw}$ *is such that* $t\ \epsilon\ A$, *then there exists* $a \in \mathtt{T}(A)$ *such that* $\mathtt{strip}\ a = t$.

The problem now is that, for a given $t\ \epsilon\ A$, there can be several such term a that corresponds to t. For instance $\mathtt{K}\ 0\ \mathtt{I}$ is a term of type N which has several possible type decorations. This is a typical *coherence* problem, needed in general to justify an overloaded notation. The following proposition solves this coherence problem, but *only* for the pure system of typed combinators without *rec*.

Proposition 15 *If* $a, a' \in \mathtt{T}(A)$ *are such that* $\mathtt{strip}\ a = \mathtt{strip}\ a'$, *then* $a\ \mathtt{conv}_A\ a'$.

Proof: We don't know any *direct* proof of this proposition. The following indirect argument is due to Streicher [29]. First, we prove unicity of a typed decoration of a raw term $t\ \epsilon\ A$ for t in normal form. The proposition results then from the normalisation theorem and the fact that $\mathtt{strip}$ preserves conversion. □

This proposition *does not hold* for the system with *rec*. Indeed, it is then possible to exhibit raw terms that have non convertible decorations, as shown by Salvesen [27]. It is thus important to work with decorated terms in this case.

6.2 On equality in the metalanguage

The quotation [18] in the introduction of this section argues that in an intuitionistic notion of model it is most fruitful to interpret equality (conversion) in the object language as definitional equality in the model. This requirement is satisfied for our formalized models. Both the intended model in 2.2 and the non-standard model in 2.3 satisfy the following remarkable definitional equalities:

$$\begin{aligned} [\![\mathtt{app}(\mathtt{app}(\mathtt{K}, a), b)]\!] &= [\![a]\!] \\ [\![\mathtt{app}(\mathtt{app}(\mathtt{app}(\mathtt{S}, g), f), a)]\!] &= [\![\mathtt{app}(\mathtt{app}(g, a), \mathtt{app}(f, a))]\!] \end{aligned}$$

The metalanguage expressions on both sides have the same normal form. As a consequence, the ALF-proofs that equality in the object language is mapped to equality in the metalanguage (theorems 1 and 3) are essentially done automatically by ALF's normalization procedure.

However, when we represent the fact that the interpretation function maps equal terms in the object language to equal elements in the model as a formal proposition in type theory, we have to replace definitional equality by 'intensional' propositional equality I:

$$A : \mathtt{Type}, x : \mathtt{T}(A), x' : \mathtt{T}(A), x\ \mathtt{conv}\ x' \vdash I(x, x')$$

The proof of this proposition is by induction on the proof of $a\ \mathtt{conv}\ a'$ and is almost immediately mechanizable, since each case of the induction is immediately reduced to a proof of reflexivity by ALF's normalization.

However, it is not the case that the *judgement*

$$A : \mathtt{Type}, x : \mathtt{T}(A), x' : \mathtt{T}(A), x\ \mathtt{conv}\ x' \vdash x = x'$$

about definitional equality is valid, since x and x' are different normal forms.

6.3 Representing algebraic notions in type theory

When representing an algebraic structure in type theory it is often the case that the appropriate notion of carrier is not that of a set but a set with an equivalence relation. For example, a *monoid* in type theory is given by a set M, an equivalence relation E, a unit element e, and a multiplication operation $*$ respecting the equivalence relation E, together with proofs that $*$ is associative (with respect to E) and that e is a left and right unit of $*$ (with respect to E).

Similarly, a *typed combinatory algebra* in type theory is given by a family of sets $M(A)$ with equivalence relations $E(A)$ indexed by types A; families of elements indexed by types $K_M(A, B)$ and $S_M(A, B, C)$ and an application operation $app_M(A, B)$ respecting the equivalence relation; together with proofs of the combinatory axioms (formulated with respect to the equivalence relation).

Algebraic structures can be formalized as *contexts* in type theory [9, 21], that is, as lists of variables and types of the form $[x_1 : \alpha_1, \ldots, x_n : \alpha_n]$. We have dependent types so that a type of a variable may depend on earlier variables. The notion of a typed combinatory algebra can hence be formalized as the following context:

$$
\begin{array}{rcl}
[M & : & (A : \mathtt{Type})Set; \\
K_M & : & (A, B : \mathtt{Type})M(A \Rightarrow B \Rightarrow A); \\
S_M & : & (A, B, C : \mathtt{Type})M((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C); \\
app_M & : & (A, B : \mathtt{Type}; M(A \Rightarrow B); M(A))M(B) \\
E & : & (A : \mathtt{Type}; M(A); M(A))Set; \\
ref_E & : & (A : \mathtt{Type}; a : M(A))E(a, a); \\
sym_E & : & (A : \mathtt{Type}; a, b : M(A); E(a, b))E(b, a); \\
trans_E & : & (A : \mathtt{Type}; a, a', a'' : M(A); E(a, a'); E(a', a''))E(a, a''); \\
appcong & : & (A, B : \mathtt{Type}; c, c' : M(A \Rightarrow B); a, a' : M(A); E(c, c'); E(a, a'))E(app_M(c, a), app_M(c', a')); \\
K_M axiom & : & (A, B : \mathtt{Type}; a : M(A); b : M(B))E(app_M(app_M(K_M, a), b), a); \\
S_M axiom & : & (A, B, C : \mathtt{Type}; g : M(A \Rightarrow B \Rightarrow C))); f : M(A \Rightarrow B); a : M(A)) \\
 & & E(app_M(app_M(app_M(\mathtt{S}, g), f), a), app_M(app_M(g, a), app_M(f, a)))]
\end{array}
$$

Here we have used the type-theoretic notation for dependent function types: $(x : \alpha)\beta[x]$ is the set of functions f which maps an object $a : \alpha$ to an object $f(a) : \beta[a]$. Set is the type of sets in the type-theoretic sense, so $(A : \mathtt{Type})Set$ is the type of $\mathtt{Type}$-indexed families of sets.

Typed combinatory algebras where E is propositional identity I are especially interesting and simple. We call them *strict* to suggest that a distinction reminiscent of that between a strict and non-strict notion in category theory. In the context formalizing the notion of a strict combinatory algebra we can omit several components:

$$
\begin{array}{rcl}
[M & : & (A : \mathtt{Type})Set; \\
K_M & : & (A, B : \mathtt{Type})M(A \Rightarrow B \Rightarrow A); \\
S_M & : & (A, B, C : \mathtt{Type})M((A \Rightarrow B \Rightarrow C) \Rightarrow (A \Rightarrow B) \Rightarrow A \Rightarrow C); \\
app_M & : & (A, B : \mathtt{Type}; M(A \Rightarrow B); M(A))M(B) \\
K_M axiom & : & (A, B : \mathtt{Type}; a : M(A); b : M(B))I(app_M(app_M(K_M, a), b), a); \\
S_M axiom & : & (A, B, C : \mathtt{Type}; g : M(A \Rightarrow B \Rightarrow C))); f : M(A \Rightarrow B); a : M(A)) \\
 & & I(app_M(app_M(app_M(\mathtt{S}, g), f), a), app_M(app_M(g, a), app_M(f, a)))]
\end{array}
$$

A particular instance of an algebraic notion (a particular monoid, a particular typed combinatory algebra, etc.) can be formalized as an *explicit substitution*, that is, as an assignment $\{x_1 := a_1; \ldots x_n := a_n\}$ of constants to variables in the appropriate context.

For example, the intended model described in 2.2 is a strict combinatory algebra formalized by the following explicit substitution:

$$
\begin{array}{rcl}
\{M & := & \mathtt{T}; \\
K_M & := & \lambda x.\lambda y.x;
\end{array}
$$

$$
\begin{aligned}
S_M &:= \lambda g.\lambda f.\lambda x.g\ x\ (f\ x);\\
app_M &:= app_M;\\
K_M axiom &:= ref;\\
S_M axiom &:= ref\}
\end{aligned}
$$

where ref is the proof of reflexivity, compare the discussion in 6.2.

Furthermore, we can formally define and instantiate the notions of homomorphism of models and of an initial model inside type theory. In particular we can build the glued model of section 2.5. Note also that this construction can be performed on any model, and not only the term model. In this way we can define abstractly the normalization function over any initial algebra.

7 Related work

Lafont [14] used glueing for proving termination and coherence theorems for categorical combinators. These results were then used to derive the evaluation mechanism of the categorical abstract machine. There is a close connection between his construction and our glueing construction (and the term "quote" is Lafont's). The fundamental difference between his work and ours is in our application to normalization and its corollaries. The difference in attitude and goals can be illustrated by the following remark, where he argues that the semantic component of his interpretation cannot directly be used for computing: 'Mais les *valeurs abstraites* de $A \to B$, avec leur composante fonctionelle, ne semblent guère "mechanisable" ' [14][page 18]. In contrast to this, we make use of the fact that these abstract values, when represented in our intuitionistic metalanguage, are indeed mechanizable. But, of course, the implementation of this metalanguage may still make use of an environment machine.

It is interesting to compare this situation with the following comments [17]: 'Of course, the fact that there is a not necessarily mechanical procedure for computing every function in the present theory of types does not require any proof at all for us, intelligent beings, who can understand the meaning of the types and the terms and recognize that the axioms and rules of inference of the theory are consonant with the intuitionistic notion of function according to which a function is the same as a rule or method.'

Related to this discussion is the following question: what kind of strategy (call-by-value, call-by-name, etc.) does the normalization algorithm extracted from these semantical arguments follow? The answer is simple: it is exactly the strategy used at the meta-level.

The technique in this paper can easily be generalized to typed λ-calculus with weak reduction, where no reduction under λ is allowed. For details we refer to the preliminary version of the present paper [8].

Berger and Schwichtenberg [5] showed how to obtain an algorithm which returns long η-normal forms for simply typed λ-calculus by inverting an interpretation function into the standard model. Berger [4] also showed how this function can be obtained from a standard normalization proof by using a modified realizability model for program extraction.

Catarina Coquand [6] constructed a similar algorithm which returns long η-normal forms for a version of the simply typed λ-calculus. Her approach is more algebraic than Berger and Schwichtenberg's. Another difference is that she inverts the interpretation function into a Kripke model. This proof has also been given a categorical reconstruction by Altenkirch, Hofmann, and Streicher [2].

Similar techniques as ours have also been considered for other purposes than normalization. Pfenning and Lee [26] considered a notion of metacircularity for the polymorphic λ-calculus and defined an 'approximately metacircular interpreter' similar to our 'intended semantics'. Mogensen [24] considered similar notions for the untyped λ-calculus intended to be used as a foundation for partial evaluation. He defined a *self-interpreter* similar to our intended semantics and a *self-reducer* similar to our normalizer. Both these papers use *higher-order abstract syntax* for representing λ-terms, whereas we use a concrete representation. A related use of glueing is de Vrijer's [10] method for getting an exact estimate of the height of the reduction tree of a term.

References

[1] T. Altenkirch, V. Gaspes, B. Nordström, and B. von Sydow. A user's guide to ALF. Draft, January 1994.

[2] T. Altenkirch, M. Hofmann, and T. Streicher. Categorical reconstruction of a reduction free normalization proof. In D. Pitt, D. E. Rydeheard, and P. Johnstone, editors, *Springer LNCS 953, Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK*, August 1995.

[3] H. P. Barendregt. *The Lambda Calculus*. North-Holland, 1984. Revised edition.

[4] U. Berger. Program extraction from normalization proofs. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht*, March 1993.

[5] U. Berger and H. Schwichtenberg. An inverse to the evaluation functional for typed λ-calculus. In *Proceedings of the 6th Annual IEEE Symposium on Logic in Computer Science, Amsterdam*, pages 203–211, July 1991.

[6] C. Coquand. From semantics to rules: a machine assisted analysis. In E. Börger, Y. Gurevich, and K. Meinke, editors, *Proceedings of CSL '93, LNCS 832*, 1993.

[7] T. Coquand. Pattern matching with dependent types. In *Proceedings of The 1992 Workshop on Types for Proofs and Programs*, June 1992.

[8] T. Coquand and P. Dybjer. Intuitionistic model constructions and normalization proofs. Preliminary Proceedings of the 1993 TYPES Workshop, Nijmegen, 1993.

[9] N. G. de Bruijn. Telescopic mappings in typed lambda calculus. *Information and Computation*, (91):189–204, 1991.

[10] R. de Vrijer. Exactly estimating functionals and strong normalization. In *Proceedings of the Koninklije Nederlandse Akademi van Wettenschappen, Series A*, volume 90, pages 479–493, 1987.

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

[12] P. Dybjer. Inductive families. *Formal Aspects of Computing*, pages 440–465, 1994.

[13] J. Hindley and J. Seldin. *Introduction to Combinators and Lambda-Calculus*. London Mathematical Society, Student Text 1. Cambridge University Press, 1986.

[14] Y. Lafont. *Logique, Categories & Machines. Implantation de Langages de Programmation guidée par la Logique Catégorique*. PhD thesis, l'Universite Paris VII, January 1988.

[15] J. Lambek and P. Scott. *Introduction to Higher Order Categorical Logic*. Cambridge University Press, 1986.

[16] P. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[17] P. Martin-Löf. An intuitionistic theory of types. Unpublished report, 1972.

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

[19] P. Martin-Löf. An intuitionistic theory of types: Predicative part. In *Logic Colloquium '73*, pages 73–118. North-Holland, 1975.

[20] P. Martin-Löf. *Intuitionistic Type Theory*. Bibliopolis, 1984.

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

[22] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[23] J. C. Mitchell and A. Scedrov. Notes on sconing and relators. In E. Börger et al., editor, *Computer Science Logic '92, Selected Papers*, pages 352–378. Springer Lecture Notes in Computer Science 702, 1993.

[24] T. Æ. Mogensen. Efficient self-interpretation in lambda calculus. *Journal of Functional Programming*, 2(3):345–364, 1992.

[25] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory: an Introduction*. Oxford University Press, 1990.

[26] F. Pfenning and P. Lee. Metacircularity in the polymorphic lambda-calculus. *Theoretical Computer Science*, 89:137–159, 1991.

[27] A. Salvesen. *On information discharging and retrieval in Martin-Löf Type Theory*. PhD thesis, University of Oslo, 1989.

[28] J. E. Stoy. *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*. The MIT Press, 1977.

[29] T. Streicher. *Correctness and Completeness of a Categorical Semantics of the Calculus of Constructions*. PhD thesis, Fakultät für Mathematik und Informatik, Universität Passau, 1988.