Intuitionistic Model Constructions and Normalization Proofs

Thierry Coquand and Peter Dybjer

June 25, 1998

Abstract

We investigate semantical normalization proofs for typed combinatory logic and weak λ-calculus. One builds a model and a function 'quote' which inverts the interpretation function. A normalization function is then obtained by composing quote with the interpretation function.

Our models are just like the intended model, except that the function space includes a syntactic component as well as a semantic one. We call this a 'glued' model because of its similarity with the glueing construction in category theory. Other basic type constructors are interpreted as in the intended model. In this way we can also treat inductively defined types such as natural numbers and Brouwer ordinals.

We also discuss how to formalize λ-terms, and show how one model construction can be used to yield normalization proofs for two different typed λ-calculi – one with explicit and one with implicit substitution.

The proofs are formalized using Martin-Löf's type theory as a meta language and mechanized using the ALF interactive proof checker. Since our meta language is intuitionistic, any normalization function is a normalization algorithm. Moreover, our algorithms can be seen as optimized versions of normalization proofs by the reducibility method, where the parts of the proof which play no role in returning a normal form are removed.

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 denotional semantics of a language and the rules of evaluation in an environment machine [17] [1].

In this paper we use this analogy to give a semantical treatment of normalization for simply typed combinators and λ-calculus with weak reduction. The method consists of building a non-standard model, and a function ('quote') which maps a semantic object to a normal term representing it.

Our approach is strongly inspired by two early papers by Martin-Löf, where he emphasized the importance of intuitionistic abstractions on the meta level and the notion of definitional equality [19] and proved normalization for his type theory by using a model construction [20].

We pursue these ideas further. In particular we wish to emphasize the following aspects of our normalization proofs:

- They are expressed as properties of normalization algorithms, rather than as the usual $\forall\exists$-propositions referring to binary reduction relations.
- The normalization algorithms are obtained by 'program extraction' from standard normalization proofs using the reducibility method. (Since any constructive proof is an algorithm it would be better to talk about optimization of a proof seen as a program.) The resulting models are simplifications of Martin-Löf's [19].

[1]The fundamental importance of this analogy was stressed by Per Martin-Löf in a recent talk about a substitution calculus.

- The model constructions can be nicely expressed in the framework of initial algebra semantics (formalized in our constructive setting). We also discuss the role of definitional equality in this context [19].
- The models can be thought of as 'glueing syntax with semantics' a technique analogous to glueing (or sconing, or Freyd cover) in category theory [16, 22]. This technique has also been used by Lafont [15] for proving a coherence result for categorical combinators.
- Syntactic properties, such as Church-Rosser, may be replaced by semantic ones, such as the property that two terms are convertible iff their semantics are equal in the glued model.
- Martin-Löf's type theory is used as a formal meta language.
- The proofs are implemented on a machine using the interactive proof checker ALF.

We first develop a proof for pure typed combinatory logic (or positive implicational calculus). Then we extend it to full propositional logic, and note that all connectives except implication are interpreted as in the intended model. Finally, we show that the proof extends directly to inductively defined types, such as the natural numbers and Brouwer ordinals.

A very similar method also works for simply typed λ-calculus with weak reduction, where no reduction under λ is allowed. Here we focus on the representation problem for λ-terms. In particular we build a glueing model from normal λ-terms and meanings, and use it for normalization of two different versions of the λ-calculus. One of these has explicit substitutions and is similar to the $\lambda\sigma$-calculus of Abadi, Cardelli, Curien, and Lévy [1]. The other is a nameless variant of the calculus used by Martin-Löf [20].

In an accompanying paper, a similar technique is used by Catarina Coquand [6] for normalization in simply typed λ-calculus with full reduction. In this case a Kripke model is used for the non-standard semantics.

2 Type theory and ALF

2.1 Martin-Löf's type theory

The formal meta language is Martin-Löf's type theory with inductive definitions and pattern matching. We use the intensional version of type theory formulated by Martin-Löf 1986, see Nordström, Petersson, and Smith [24].

The core of this language is the *theory of logical types* (*Martin-Löf's logical framework*). This is a dependently typed $\lambda\beta\eta$-calculus with a base type *Set* and a base type A for each object $A : Set$. We use the notation

$$(x_0 : \alpha_0; \ldots ; x_n : \alpha_n)\alpha$$

for the type of n-ary functions[2];

$$[x_0, \ldots , x_n]a$$

for n-ary function abstraction; and

$$a(a_1, \ldots , a_n)$$

for n-ary function application.

We then use this framework for defining new sets and families of sets (*Martin-Löf's set theory*). These are defined by their constructors (or *introduction rules*). The rules follow Dybjer [11, 12], who gave natural deduction formulation of a class of admissible such *inductive definitions* in type theory. (See also Coquand and Paulin [8] for similar ideas in the context of the Calculus of Constructions.) This includes all standard set formers of Martin-Löf's type theory. Here we also introduce sets of (object language) types, families of sets of terms, families of conversion relations, etc. These are ordinary inductive definitions. In addition we need to define non-standard ordinals in the glued model by a generalized inductive definition.

[2] An x_i which is not referred to by later types may be omitted.

To highlight the relationship between corresponding notions on the meta level and the object level we use certain notational conventions illustrated by the following table:

meta language	object language
$\rightarrow$	$\dot{\rightarrow}$
λ	$\dot{\lambda}$
app	$\mathtt{app}$

There are at least two notions of equality in type theory: *definitional equality*, written $a = b : A$, and *intensional equality*, written $I(A, a, b)$ (we often drop A in either case). Definitional equality is decidable and expressed by the equality *judgement*. Two terms are definitionally equal iff they are convertible iff they can be reduced to the same normal form by unfolding (possibly recursive) definitions. Intensional equality is expressed on the *propositional* level; it is a binary relation which is inductively defined by the reflexivity rule

$$ref : (A : Set; a : A)I(A, a, a)$$

If we have $a = b : A$, then $ref(a)$ is a proof of $I(A, a, b)$ by substitutivity of definitional equality. Conversely, if we have a closed proof of $I(A, a, b)$, where $a, b : A$ are closed terms, then $a = b : A$. This can be justified by appealing to the semantics of type theory: a closed proof of $I(A, a, b)$ should reduce to a closed canonical proof $ref(u)$ with $u = a = b : A$.

Furthermore, we introduce new functions by *pattern matching* following the ideas of Coquand [7]. This facility provides both a convenient notation and a useful generalization of the standard elimination rules or primitive recursive schemata [12] that can be derived from the introduction rules for a set.

Firstly, it allows the definition of functions by case analysis on several arguments simultaneously and uses a criterion that recursive calls must refer to *structurally smaller* arguments to ensure termination.

Secondly, unification is used to generate possible cases. This entails a strengthening of case analysis for inductively defined families. An example is that the proof of

$$peano4 : (I(0, 1))\emptyset$$

now follows directly by pattern matching. The introduction rule for equality is reflexivity, and since this rule cannot be unified with $I(0, 1)$ no cases are generated.

In our proofs we have used pattern matching in a non-essential way, and we shall indicate below how these uses can be reduced to standard primitive recursive schemata.

2.2 ALF - an interactive proof checker

We have implemented our proofs using the interactive proof checker ALF (Another Logical Framework) developed by Augustsson, Coquand, Magnusson, and Nordström. ALF is an implementation of Martin-Löf's logical framework. It is easy to introduce new constants and computation rules, for example, the formation and introduction rules for new sets and the typing and computation rules for functions defined by pattern matching. ALF implements the generation of cases described by Coquand [7], but the checks that recursive calls refer to structurally smaller arguments and that formation and introduction rules have the correct form have to be done manually.

The terms we present are edited versions of terms generated by the machine. The main difference is the use of implicit arguments (when they are clear from the context; this facility is not part of the ALF implementation yet) and overloading for making the notation more readable.

3 Typed combinatory logic

We first discuss pure typed combinatory logic, which by the Curry-Howard identification corresponds to a Hilbert-style axiomatization of positive implicational calculus. We give its syntax, its intended semantics, a normalization algorithm, and a correctness proof for this algorithm. We discuss algebraic and categorical aspects, and also how the algorithm can be extracted from a more standard proof using the reducibility method.

We then extend the approach to new type constructors which yield full intuitionistic propositional calculus. Finally, we consider inductively defined types, such as natural numbers and Brouwer ordinals.

3.1 Pure typed combinatory logic

3.1.1 Syntax

The identification of propositions and types strongly suggests to choose a formulation à la Church rather than à la Curry [3]. In the the former terms appear as proof objects of a Curry-Howard interpretation of the positive implicational calculus. The formulation à la Curry introduces redundant information.

We begin by defining the set of types inductively. The formation and introduction rules are

$$\texttt{Type}^3 : Set$$

$$\dot{\rightarrow} \quad : \quad (\texttt{Type}; \texttt{Type})\texttt{Type}$$

We introduce more types below; at this point the reader may wish to add an uninterpreted base type $o : \texttt{Type}$.

The terms form an inductively defined family $\texttt{T}$ of sets indexed by types. The formation and introduction rules are

$$\texttt{T} : (\texttt{Type})Set$$

$$\begin{aligned}
\texttt{K} \quad &: \quad (A, B : \texttt{Type})\texttt{T}(A \dot{\rightarrow} B \dot{\rightarrow} A) \\
\texttt{S} \quad &: \quad (A, B, C : \texttt{Type})\texttt{T}((A \dot{\rightarrow} B \dot{\rightarrow} C) \dot{\rightarrow} (A \dot{\rightarrow} B) \dot{\rightarrow} A \dot{\rightarrow} C) \\
\texttt{app} \quad &: \quad (A, B : \texttt{Type}; \texttt{T}(A \dot{\rightarrow} B); \texttt{T}(A))\texttt{T}(B)
\end{aligned}$$

3.1.2 Intended semantics

It is now possible to relate object language and meta language notions by giving the *intended* semantics. This is Tarski style semantics with intuitionistic notions on the meta level. What may be a little confusing at first here is that both meta level and object level are formalized: it is an interpreter (for terms of base type) written in ALF.

We define the interpretation of $\texttt{Type}$ by recursion:

$$[\![\;]\!] : (A : \texttt{Type})Set$$

$$[\![A \dot{\rightarrow} B]\!] \quad = \quad [\![A]\!] \rightarrow [\![B]\!]$$

The interpretation of $\texttt{T}$ is also by recursion:

$$[\![\;]\!] : (A : \texttt{Type}; a : \texttt{T}(A))[\![A]\!]$$

$$\begin{aligned}
[\![\texttt{K}]\!] \quad &= \quad \lambda([x]\lambda([y]x)) \\
[\![\texttt{S}]\!] \quad &= \quad \lambda([g]\lambda([f]\lambda([x]app(app(g, x), app(f, x))))) \\
[\![\texttt{app}(f, a)]\!] \quad &= \quad app([\![f]\!], [\![a]\!])
\end{aligned}$$

Note that this is structural recursion on an inductively defined family of sets.

Martin-Löf [19] discusses an intuitionistic notion of model and argues that equality (conversion) in the object language should be interpreted as definitional equality in the model. This requirement is satisfied for our (formalized) intended model in the following sense.

[3] We use the word 'type' and the formal name `Type` for 'type expression'. It would be less convenient but more consistent with the meta language to talk about 'set expression' and use the formal name `Set`.

Firstly, we have the remarkable definitional equalities:

$$\begin{aligned}
[\![\texttt{app}(\texttt{app}(\texttt{K}, x), y)]\!] &= [\![x]\!] \\
[\![\texttt{app}(\texttt{app}(\texttt{app}(\texttt{S}, x), y), z)]\!] &= [\![\texttt{app}(\texttt{app}(x, z), \texttt{app}(y, z))]\!]
\end{aligned}$$

The meta language expressions on both sides have the same normal form.

Moreover, we can introduce conversion as a family of inductively defined relations indexed by the types:

$$\texttt{I}(,) : (A : \texttt{Type}; a, a' : \texttt{T}(A))Set$$

$$\begin{aligned}
\texttt{convK} &: (A, B : \texttt{Type}; a : \texttt{T}(A); b : \texttt{T}(B))\texttt{I}(\texttt{app}(\texttt{app}(K, a), b), a) \\
\texttt{convS} &: (A, B, C : \texttt{Type}; g : \texttt{T}(A \dot{\to} B \dot{\to} C); f : \texttt{T}(A \dot{\to} B); a : \texttt{T}(A)) \\
&\quad\; \texttt{I}(\texttt{app}(\texttt{app}(\texttt{app}(\texttt{S}, g), f), a), \texttt{app}(\texttt{app}(g, a), \texttt{app}(f, a))) \\
\texttt{convapp} &: (A, B : \texttt{Type}; c, c' : \texttt{T}(A \dot{\to} B); a, a' : \texttt{T}(A); \texttt{I}(c, c'); \texttt{I}(a, a'))\texttt{I}(\texttt{app}(c, a), \texttt{app}(c', a')) \\
\texttt{convref} &: (A : \texttt{Type}; a : \texttt{T}(A))\texttt{I}(a, a) \\
\texttt{convsym} &: (A : \texttt{Type}; a, b : \texttt{T}(A); \texttt{I}(a, b))\texttt{I}(b, a) \\
\texttt{convtrans} &: (A : \texttt{Type}; a, a', a'' : \texttt{T}(A); \texttt{I}(a, a'); \texttt{I}(a', a''))\texttt{I}(a, a'')
\end{aligned}$$

We can prove by induction on $\texttt{I}(,)$ that

$$(A : \texttt{Type}; a, a' : \texttt{T}(A); \texttt{I}(a, a'))I([\![a]\!], [\![a']\!])$$

This proof is almost immediately mechanizable since most of the work is done by ALF's normalization. By the discussion of equality in section 2.1, we know that we can reexpress this proposition as the meta level statement that if $\texttt{I}(a, a')$ (under no assumptions), then the definitional equality $[\![a]\!] = [\![a']\!]$ follows.

3.1.3 Normalization algorithm

It is impossible to invert the interpretation function for the intended model, but if we enrich the interpretation of $\dot{\to}$ so that it has both a syntactic and a semantic component this becomes possible.

First we define a new interpretation function for types[4] [5]

$$[\![A \dot{\to} B]\!] = \texttt{T}(A \dot{\to} B) \times ([\![A]\!] \to [\![B]\!])$$

From this model we can retrieve normal forms:

$$\mathbf{q} : (A : \texttt{Type}; [\![A]\!])\texttt{T}(A)$$

$$\mathbf{q}_{A \dot{\to} B}(\langle c, f \rangle) = c$$

This is a first simple use of pattern matching with dependent types: we analyze both the first (implicit) and the second argument (which depends on the first one) simultaneously. Here it is straightforward to transform this definition into a standard primitive recursive schema (using higher types):

$$\mathbf{q}_{A \dot{\to} B}(p) = \mathit{fst}(p)$$

The interpretation of terms becomes:

$$[\![\ ]\!] : (A : \texttt{Type}; \texttt{T}(A))[\![A]\!]$$

[4] Alternatively, we could have introduced sets of normal terms $\texttt{Nt}(A)$ and used them as the syntactic component of the model instead. In that way we would formally ensure that the retrieved term is normal. This may be a technical advantage: for example one can use essentially the same set of normal terms for different kinds of λ-calculi, see section 4.2.1.

[5] Base types are interpreted syntactically: $[\![o]\!] = \texttt{T}(o)$

$$
\begin{aligned}
[\![\mathtt{K}]\!] &= \langle \mathtt{K}, \lambda([p]\langle \mathtt{app}(\mathtt{K}, \mathbf{q}(p)), \lambda([q]p)\rangle\rangle \\
[\![\mathtt{S}]\!] &= \langle \mathtt{S}, \lambda([p]\langle \mathtt{app}(\mathtt{S}, \mathbf{q}(p)), \lambda([q]\langle \mathtt{app}(\mathtt{app}(\mathtt{S}, \mathbf{q}(p)), \mathbf{q}(q)), \lambda([r]\mathit{app}_M(\mathit{app}_M(p,r), \mathit{app}_M(q,r)))\rangle)\rangle)\rangle \\
[\![\mathtt{app}(f,a)]\!] &= \mathit{app}_M([\![f]\!], [\![a]\!])
\end{aligned}
$$

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

$$\mathit{app}_M : (A, B : \mathtt{Type}; [\![A \dot{\rightarrow} B]\!]; [\![A]\!])[\![B]\!]$$

$$\mathit{app}_M(\langle c, f\rangle, q) \quad = \quad \mathit{app}(f, q)$$

Finally, the normal form is extracted:

$$\mathbf{nf} : (A : \mathtt{Type}; \mathtt{T}(A))\mathtt{T}(A)$$

$$\mathbf{nf}(a) \quad = \quad \mathbf{q}([\![a]\!])$$

Also in this model we have the definitional equalities:

$$
\begin{aligned}
[\![\mathtt{app}(\mathtt{app}(\mathtt{K}, x), y)]\!] &= [\![x]\!] \\
[\![\mathtt{app}(\mathtt{app}(\mathtt{app}(\mathtt{S}, x), y), z)]\!] &= [\![\mathtt{app}(\mathtt{app}(x, z), \mathtt{app}(y, z))]\!]
\end{aligned}
$$

and

$$(A : \mathtt{Type}; a, a' : \mathtt{T}(A); \mathtt{I}(a, a'))I([\![a]\!], [\![a']\!])$$

and hence

$$(A : \mathtt{Type}; a, a' : \mathtt{T}(A); \mathtt{I}(a, a'))I(\mathbf{nf}(a), \mathbf{nf}(a'))$$

so that the rules of conversion are sound for the normal form semantics.

From this model we can also easily extract the model of normal terms.

As pointed out to us by Thorsten Altenkirch, this program can be translated into ML, 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);
```

3.1.4 Normalization proof

We shall prove that our normalization algorithm is correct in the sense that

$$(A : \mathtt{Type}; a : \mathtt{T}(A))\mathtt{I}(a, \mathbf{nf}(a)).$$

Together with the fact that convertible terms have equal normal forms, which we proved above, and the fact that intensional equality is decidable, this yields a decision algorithm for convertibility.

According to this view correctness amounts to showing that two *syntactic* notions are equivalent: convertibility and equality of normal forms. But there is also a weaker but more fundamental form of *semantic* correctness: the normalization algorithm is correct since it preserves semantical equality. (This may be either intensional or extensional equality of elements in the intended model.)

Normalization yields an incomplete decision procedure for semantic equality. This is simply because weak conversion fails to distinguish between ξ and η-convertible terms and these conversions preserve the intended semantics. However it is complete for the glued semantics that we present below.

The correctness proof uses a construction closely related to glueing from categorical logic. Recall that function spaces in the model have a syntactic and a semantic component. We shall require that these components are coherent with each other: they are correctly 'glued' together in the sense that quotation commutes with application. This point will be clarified and expanded in the next section.

We introduce the property *Gl* to formalize this:

$$Gl : (A : \mathtt{Type}; [\![A]\!])Set$$

$$Gl_{A \dot{\rightarrow} B}(\langle c, f \rangle) \quad = \quad \Pi([\![A]\!], [p](Gl_A(p) \rightarrow (Gl_B(app(f,p)) \times (\mathtt{I}(\mathtt{app}(c, \mathbf{q}(p)), \mathbf{q}(app(f,p)))))))$$

We can now show that each term is interpreted as a glued value:

$$gl : (A : \mathtt{Type}; a : \mathtt{T}(A))\, Gl([\![a]\!])$$

$$\begin{aligned} gl(\mathtt{K}) &= \lambda([p]\lambda([x]\langle\lambda([q]\lambda([y]\langle x, \mathtt{convK}\rangle)), \mathtt{convref}\rangle)) \\ gl(\mathtt{S}) &= (long\ term) \\ gl(\mathtt{app}(c,a)) &= app_{Gl}([\![c]\!], gl(c), [\![a]\!], gl(a)) \end{aligned}$$

where

$$app_{Gl} : (A, B : \mathtt{Type}; q : [\![A \dot{\rightarrow} B]\!]; y : Gl(q); p : [\![A]\!]; x : Gl(p))\, Gl(app_M(q,p))$$

$$app_{Gl}(q, y, p, x) = fst(app(app(y,p), x))$$

From this it follows easily by $\mathtt{T}$-recursion that

$$(A : \mathtt{Type}; a : \mathtt{T}(A))\mathtt{I}(a, \mathbf{nf}(a))$$

This is an *external* proof of correctness in the sense that we first write the program **nf**, and then we make a logical comment on it. Note that in this way we do $\mathtt{T}$-recursion (induction) *three* times: one for the program, one for glueing and one for convertibility. Alternatively, we could directly prove by *one* $\mathtt{T}$-recursion that

$$(A : \mathtt{Type}; a : \mathtt{T}(A))\Sigma([\![A]\!], [p]\, Gl(p) \times (\mathtt{I}(a, \mathbf{q}(p))))$$

This would be an *integrated* proof of correctness. If we only care about the normal form, however, it contains parts which are irrelevant. If the program is optimized by removing these parts we get **nf** back.

3.1.5 Algebraic view of the normalization proof

One can present the normalization proof algebraically by observing that

- the algebra of combinatory terms is initial among typed combinatory algebras;
- the model of glued values is another typed combinatory algebra;
- the interpretation function mapping each term to a glued element (that is a value in the model and a proof that it is glued) is the unique homomorphism from the initial algebra;

- the quote function (restricted to glued values) is a homomorphism back to the initial algebra (but it is not a morphism on the original algebra of combinatory terms).

Hence the normalization function is a homomorphism on the initial algebra. Hence it is (extensionally) equal to the identity. Since equality on terms is convertibility, this means that the normalization function preserves convertibility.

We shall now formalize these points in type theory using the fact that we can express model notions as *contexts* and instances of such notions as *explicit substitutions*.

Firstly, the *signature* Σ_{TCL} of typed combinatory algebras is represented by the following context:

$$\begin{array}{rcl} [M & : & (A : \mathtt{Type})Set; \\ K_M & : & (A, B : \mathtt{Type})M(A \dot{\rightarrow} B \dot{\rightarrow} A); \\ S_M & : & (A, B, C : \mathtt{Type})M((A \dot{\rightarrow} B \dot{\rightarrow} C) \dot{\rightarrow} (A \dot{\rightarrow} B) \dot{\rightarrow} A \dot{\rightarrow} C); \\ app_M & : & (A, B : \mathtt{Type}; M(A \dot{\rightarrow} B); M(A))M(B)] \end{array}$$

The intended model of these axioms is represented by the following substitution:

$$\begin{array}{rcl} [M & := & [\![\ ]\!]; \\ K_M & := & [A, B]\lambda([x]\lambda([y]x)); \\ S_M & := & [A, B, C]\lambda([g]\lambda([f]\lambda([a]app(app(g, a), app(f, a))))); \\ app_M & := & app] \end{array}$$

As for the representation of the axioms for combinators we have two choices. The first is to follow Martin-Löf and let equality in the model be intensional:

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

As discussed above the intended model immediately satisfies these axioms. However, the term model does not because there we need to *reinterpret* equality as convertibility, and quotient formation is not a set forming operation in type theory.

So we need a second notion which includes an explicit uninterpreted equivalence relation as part of the structure:

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

We shall also need the notion of a *homomorphism* of typed combinatory algebras in the second sense. This a family of functions indexed by the types, which preserve the equivalence relation and the operations.

We can prove that the algebra of combinatory terms under convertibility is initial among typed combinatory algebras (in the second sense), since it has a unique homomorphism (up to extensional equality) to any other. The proof is the standard one interpreted in our constructive setting.

In the context of an arbitrary typed combinatory algebras (in the second sense), a homomorphism h is defined by structural recursion on combinatory terms:

$$\begin{array}{rcl} h(\mathtt{K}) & = & K_M \\ h(\mathtt{S}) & = & S_M \\ h(\mathtt{app}(c,a)) & = & app_M(h(c), h(a)) \end{array}$$

The proof that h really is a homomorphism is direct: each rule of conversion is mapped into an axiom for combinatory algebras; that h commutes with the operations is immediate from the definition.

The uniqueness proof is by induction on terms. Assume that h' is any homomorphism from the initial algebra, that is,

$$\begin{array}{l} E(h'(\mathtt{K}), K_M) \\ E(h'(\mathtt{S}), S_M) \\ E(h'(\mathtt{app}(c,a)), app_M(h'(c), h'(a))) \end{array}$$

and it follows that h and h' are extensionally equal:

$$(A : \mathtt{Type}; a, a' : \mathtt{T}(A); \mathtt{I}(a, a'))E(h(a), h'(a'))$$

Furthermore, the model of glued values is given by the following substitution

$$[M := G; K_M := K_G; S_M := S_G; app_M := app_G; K_M axiom := K_G axiom; S_M axiom := S_G axiom]$$

where

$$\begin{array}{rcl} G(A) & = & \Sigma p : [\![A]\!].Gl(p) \\ K_G & = & \langle [\![\mathtt{K}]\!], gl(\mathtt{K})\rangle \\ S_G & = & \langle [\![\mathtt{S}]\!], gl(\mathtt{S})\rangle \\ app_G(\langle q,y\rangle, \langle p,x\rangle) & = & app_{Gl}(q,y,p,x) \\ K_G axiom(\langle p,x\rangle, \langle q,y\rangle) & = & ref(\langle p,x\rangle) \\ S_G axiom(\langle p,x\rangle, \langle q,y\rangle, \langle r,z\rangle) & = & ref(\ldots) \end{array}$$

which is a combinatory algebra in the first sense.

The quote homomorphism is just $fst \circ fst$. We need to prove

$$\begin{array}{l} \mathtt{I}(\mathbf{q}(K_M), \mathtt{K}) \\ \mathtt{I}(\mathbf{q}(S_M), \mathtt{S}) \\ \mathtt{I}(\mathbf{q}(app_M(q,p)), \mathtt{app}(\mathbf{q}(q), \mathbf{q}(p))) \end{array}$$

The first two are immediate and the third follows by definition of a glued value. Indeed, the glued values are precisely those for which the third conversion holds. So all the work is in showing that we have a typed combinatory algebra of glued values.

This use of initial algebra semantics is similar to its use for structuring compiler correctness proofs, see for example Thatcher, Wagner, and Wright [28].

3.1.6 Categorical glueing

In categorical glueing [16, 22] one starts with a functor $\mathtt{T} : \mathcal{C} \to \mathcal{S}$. The glueing (or sconing) construction is a new category, which has as objects arrows

$$\mathbf{q} : X \to \mathtt{T}(A)$$

of $\mathcal{S}$ and as arrows pairs $\langle c, f\rangle$, such that

$$\mathtt{T}(c) \circ \mathbf{q} = \mathbf{q}' \circ f.$$

The Freyd cover construction is a special case of this.

We now see the similarity between the interpretation of function spaces in our glued model and the interpretation of hom-sets in categorical glueing: the commuting square states a similar requirement to the requirement that quote commutes with application.

3.1.7 Optimization of a standard normalization proof

We shall compare our proof with Martin-Löf's proof of normalization for a version of intuitionistic type theory [20] (adapted to the the simple case of typed combinatory logic). This proof is also expressed as a model construction, but is standard in the sense that it defines a reducibility predicate à la Tait. Another proof of normalization for typed combinators (also using Tait-reducibility) was implemented in ALF by Gaspes and Smith [13], but the relationship between this proof and our extracted algorithm seems somewhat less direct.

We shall here show that we can optimize this proof too and again extract **nf**! When formalizing an informal proof one always has to make explicit certain choices which were left implicit. Here we have chosen a representation which makes the optimization process as simple as possible.

So Martin-Löf [20] defined the meaning [6] of a type A relative to a term a is a triple consisting of a normal term, a proof that it is reducible (which is a property of terms corresponding to the glueing property of values) and a proof that it is normalizable:

$$[\![\ ]\!] : (A : \mathtt{Type}; a : \mathtt{T}(A))Set$$

$$[\![A, a]\!] \quad = \quad \Sigma(Red(A), [ap]\mathtt{I}(a, \mathbf{q}(ap)))$$

The first two components are put together into a pair

$$\begin{aligned} Red &: (A : \mathtt{Type})Set \\ \mathbf{q} &: (A : \mathtt{Type}; xp : Red(A))\mathtt{T}(A) \end{aligned}$$

$$\begin{aligned} Red(A \dot{\rightarrow} B) &= \Sigma(\mathtt{T}(A \dot{\rightarrow} B), [c]\Pi(Red(A), [ap]\Sigma(Red(B), [bp]\mathtt{I}(\mathtt{app}(c, \mathbf{q}(ap)), \mathbf{q}(bp))))) \\ \mathbf{q}_{A \dot{\rightarrow} B}(\langle c, f\rangle) &= c \end{aligned}$$

The integrated normalization algorithm (proof) can now be defined by

$$[\![\ ]\!] : (A : \mathtt{Type}; a : \mathtt{T}(A))[\![A, a]\!]$$

$$\begin{aligned} [\![\mathtt{K}]\!] &= \langle\langle \mathtt{K}, \lambda([xp]\langle\langle \mathtt{app}(\mathtt{K}, \mathbf{q}(xp)), \lambda([yp]\langle xp, \mathtt{convK}\rangle)\rangle, \mathtt{convref}\rangle)\rangle, \mathtt{convref}\rangle \\ [\![\mathtt{S}]\!] &= (\textit{large term}) \\ [\![\mathtt{app}(f, a)]\!] &= app_M([\![f]\!], [\![a]\!]) \end{aligned}$$

where we have used the following application operator in the model

$$app_M : (A, B : \mathtt{Type}; c : \mathtt{T}(A \dot{\rightarrow} B)); a : \mathtt{T}(A); p : [\![A \dot{\rightarrow} B, c]\!]; q : [\![A, a]\!])[\![B, \mathtt{app}(c, a)]\!]$$

$$app_M(p, q) \quad = \quad \langle fst(app_R(p, q)), \mathtt{convtrans}(\mathtt{convapp}(snd(app_R(p, q))))\rangle$$

where

$$app_R(p, q) = app(snd(fst(p)), fst(q))$$

The connection with the optimized algorithm should now be apparent: just remove all proof object for convertibility and simplify the types accordingly. This process could be made precise by using a suitable formalism for removing redundant information such as checked quantifiers or subsets. It would also be interesting to investigate automatic removal of redundant information along the lines of Takayama [26].

Berger [4] has provided a related analysis for a strong normalization proof of the typed λ-calculus, and shown that one gets the normalization algorithm of Berger and Schwichtenberg [5]. He uses an alternative framework and explains program extraction in terms of modified realizability. Only the predicate logic part of the proof, and not the parts involving induction, is treated explicitly.

[6] This is a model construction in a somewhat different sense than before, since $[\![\ ]\!]$ here is indexed by a term as well as a type.

3.2 Propositional calculus

3.2.1 Syntax

We now extend our object language with type formers corresponding to the logical constants to get full intuitionistic propositional calculus:

$$\begin{array}{rcl}
\dot{\emptyset} & : & \mathtt{Type} \\
\dot{1} & : & \mathtt{Type} \\
\dot{\times} & : & (\mathtt{Type};\mathtt{Type})\mathtt{Type} \\
\dot{+} & : & (\mathtt{Type};\mathtt{Type})\mathtt{Type}
\end{array}$$

There are also new term constructors corresponding to the introduction and elimination rules for propositional calculus [7]

$$\begin{array}{rcl}
\mathtt{case0} & : & (C:\mathtt{Type})\mathtt{T}(\dot{\emptyset}\dot{\to}C) \\
\mathtt{<>} & : & \mathtt{T}(\dot{1}) \\
\mathtt{inl} & : & (A,B:\mathtt{Type};\mathtt{T}(A))\mathtt{T}(A\dot{+}B) \\
\mathtt{inr} & : & (A,B:\mathtt{Type};\mathtt{T}(B))\mathtt{T}(A\dot{+}B) \\
\mathtt{case} & : & (A,B,C:\mathtt{Type};\mathtt{T}(A\dot{\to}C);\mathtt{T}(B\dot{\to}C))\mathtt{T}(A\dot{+}B\dot{\to}C) \\
\mathtt{<\,,\,>} & : & (A,B:\mathtt{Type};\mathtt{T}(A);\mathtt{T}(B))\mathtt{T}(A\dot{\times}B) \\
\mathtt{fst} & : & (A,B:\mathtt{Type};\mathtt{T}(A\dot{\times}B))\mathtt{T}(A) \\
\mathtt{snd} & : & (A,B:\mathtt{Type};\mathtt{T}(A\dot{\times}B))\mathtt{T}(B)
\end{array}$$

3.2.2 Intended semantics

The intended semantics is for types

$$\begin{array}{rcl}
[\![\dot{\emptyset}]\!] & = & \emptyset \\
[\![\dot{1}]\!] & = & 1 \\
[\![A\dot{+}B]\!] & = & [\![A]\!]+[\![B]\!] \\
[\![A\dot{\times}B]\!] & = & [\![A]\!]\times[\![B]\!]
\end{array}$$

and terms

$$\begin{array}{rcl}
[\![\mathtt{case0}]\!] & = & \lambda([z]\mathit{case}_0(z)) \\
[\![\mathtt{<>}]\!] & = & \langle\rangle \\
[\![\mathtt{inl}(a)]\!] & = & \mathit{inl}([\![a]\!]) \\
[\![\mathtt{inr}(b)]\!] & = & \mathit{inr}([\![b]\!]) \\
[\![\mathtt{case}(d,e)]\!] & = & \lambda([z]\mathit{case}([x]\mathit{app}([\![d]\!],x),[y]\mathit{app}([\![e]\!],y),z)) \\
[\![\mathtt{<}a,b\mathtt{>}]\!] & = & \langle[\![a]\!],[\![b]\!]\rangle \\
[\![\mathtt{fst}(a)]\!] & = & \mathit{fst}([\![a]\!]) \\
[\![\mathtt{snd}(a)]\!] & = & \mathit{snd}([\![a]\!])
\end{array}$$

From the intended semantics we immediately get a consistency proof

$$(a:\mathtt{T}(\dot{\emptyset}))\emptyset$$

which is nothing but the term interpretation function specialized to $\dot{\emptyset}$.

[7]To make things simple we sometimes give inference rules instead of axioms; it would be easy to modify the approach to deal with a with axioms only.

It is a matter of debate whether or not this can be seen as an internalization of the discussion in Martin-Löf [21] on simple-minded versus metamathematical consistency. On the one hand, it is quite tempting to look at the opposition object (syntactic) level versus meta (semantic) level (compare Tarski [27]) in the formalisation as the counterpart to the opposition syntax versus semantics according to the meaning explanations of Martin-Löf [21]. On the other hand, it can be argued that the real semantics contains completely different dimensions. For instance, the real semantics has to do with the way that we use the language, and this notion of 'use' is not captured by our formalisation.

We next extend the definition of conversion with the following rules

$$
\begin{aligned}
&\mathtt{I}(\mathtt{app}(\mathtt{case}(d,e),\mathtt{inl}(a)),\mathtt{app}(d,a))\\
&\mathtt{I}(\mathtt{app}(\mathtt{case}(d,e),\mathtt{inr}(a)),\mathtt{app}(e,b))\\
&\mathtt{I}(\mathtt{fst}(\texttt{<}a,b\texttt{>}),a)\\
&\mathtt{I}(\mathtt{snd}(\texttt{<}a,b\texttt{>}),b)
\end{aligned}
$$

together with congruence rules for the new term constructors.

3.2.3 Normalization algorithm

We get an enriched model which can be used for normalization by interpreting the new type formers as in the intended semantics. We also extend the definition of quote:

$$
\begin{array}{rcl}
\mathbf{q}_{\dot{1}}(\langle\rangle) & = & \texttt{<>}\\
\mathbf{q}_{A\dot{+}B}(\mathit{inl}(p)) & = & \mathtt{inl}(\mathbf{q}_A(p))\\
\mathbf{q}_{A\dot{+}B}(\mathit{inr}(q)) & = & \mathtt{inr}(\mathbf{q}_B(q))\\
\mathbf{q}_{A\dot{\times}B}(\langle p,q\rangle) & = & \texttt{<}\mathbf{q}_A(p),\mathbf{q}_B(q)\texttt{>}
\end{array}
$$

This is another application of pattern matching with dependent types, and here we really get a more compact definition. For example, there is no clause for $\dot{\emptyset}$, since there is no constructor for $\emptyset$. But again, it is easy to replace this by standard primitive recursive schemata for dependent types if one defines an auxiliary function for each of the syntactic type formers.

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).

$$
\begin{array}{rcl}
[\![\mathtt{case0}]\!] & = & \langle \mathtt{case0}, \lambda([z]\mathit{case}_0(z))\rangle\\
[\![\mathtt{case}(d,e)]\!] & = & \langle \mathtt{case}(\mathbf{q}([\![d]\!]),\mathbf{q}([\![e]\!])), \lambda([z]\mathit{case}([x]\mathit{app}_M([\![d]\!],x),[y]\mathit{app}_M([\![e]\!],y),z))\rangle
\end{array}
$$

3.2.4 Normalization proof

The definition of Gl can also be defined by pattern matching:

$$
\begin{array}{rcl}
Gl_{\dot{1}}(\langle\rangle) & = & 1\\
Gl_{A\dot{+}B}(\mathit{inl}(p)) & = & Gl_A(p)\\
Gl_{A\dot{+}B}(\mathit{inr}(q)) & = & Gl_B(q)\\
Gl_{A\dot{\times}B}(\langle p,q\rangle) & = & Gl_A(p)\times Gl_B(q)
\end{array}
$$

and the normalization proof extends as well.

3.3 Inductively defined types

3.3.1 Syntax

We finally introduce natural numbers and Brouwer ordinals:

$$
\begin{array}{rcl}
\dot{\mathtt{N}} & : & \mathtt{Type}\\
\dot{\mathcal{O}} & : & \mathtt{Type}
\end{array}
$$

The new constructors for terms are

$$
\begin{array}{rcl}
\mathtt{0} & : & \mathtt{T}(\mathtt{N}) \\
\mathtt{s} & : & (\mathtt{T}(\mathtt{N}))\mathtt{T}(\mathtt{N}) \\
\mathtt{rec} & : & (C:\mathtt{Type};\mathtt{T}(C);\mathtt{T}(\mathtt{N}\dot{\to}C\dot{\to}C))\mathtt{T}(\mathtt{N}\dot{\to}C) \\
\mathtt{0} & : & \mathtt{T}(\dot{\mathcal{O}}) \\
\mathtt{sup} & : & (\mathtt{T}(\mathtt{N}\dot{\to}\dot{\mathcal{O}}))\mathtt{T}(\dot{\mathcal{O}}) \\
\mathtt{ordrec} & : & (C:\mathtt{Type};\mathtt{T}(C);\mathtt{T}((\mathtt{N}\dot{\to}\dot{\mathcal{O}})\dot{\to}(\mathtt{N}\dot{\to}C)\dot{\to}C))\mathtt{T}(\dot{\mathcal{O}}\dot{\to}C)
\end{array}
$$

3.3.2 Intended semantics

$$
\begin{array}{rcl}
[\![\mathtt{N}]\!] & = & N \\
[\![\dot{\mathcal{O}}]\!] & = & \mathcal{O}
\end{array}
$$

$$
\begin{array}{rcl}
[\![\mathtt{0}]\!] & = & 0 \\
[\![\mathtt{s}(a)]\!] & = & s([\![a]\!]) \\
[\![\mathtt{rec}(d,e)]\!] & = & \lambda([z]rec([\![d]\!],[x,y]app(app([\![e]\!],x),y),z)) \\
[\![\mathtt{0}]\!] & = & 0 \\
[\![\mathtt{sup}(b)]\!] & = & sup([x]app([\![b]\!],x)) \\
[\![\mathtt{ordrec}(d,e)]\!] & = & \lambda([z]ordrec([\![d]\!],[xf,yf]app(app([\![e]\!],\lambda(xf)),\lambda(yf)),z))
\end{array}
$$

The new conversion rules are

$$
\begin{array}{l}
\mathtt{I}(\mathtt{app}(\mathtt{rec}(d,e),\mathtt{0}),d) \\
\mathtt{I}(\mathtt{app}(\mathtt{rec}(d,e),\mathtt{s}(a)),\mathtt{app}(\mathtt{app}(e,a),\mathtt{app}(\mathtt{rec}(d,e),a))) \\
\mathtt{I}(\mathtt{app}(\mathtt{ordrec}(d,e),\mathtt{0}),d) \\
\mathtt{I}(\mathtt{app}(\mathtt{ordrec}(d,e),\mathtt{sup}(b)),\mathtt{app}(\mathtt{app}(e,b),\mathtt{ordrec}(d,e)\circ b))
\end{array}
$$

together with congruence rules for the new term constructors. Here we have used an auxiliary syntactic binary composition operator

$$\circ : (A,B,C:\mathtt{Type};\mathtt{T}(B\dot{\to}C);\mathtt{T}(A\dot{\to}B))\mathtt{T}(A\dot{\to}C)$$

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

We note that it is sufficient to build the 'intended model' to prove equational consistency

$$(\mathtt{I}(\mathtt{0},\mathtt{s}(\mathtt{0})))\emptyset$$

However, to prove for example that the constructor s is one-to-one for convertibility we need the normalization proof.

3.3.3 Normalization algorithm

The enriched model for inductively defined types is obtained by the same principle. Only constructors with functional arguments need to be reinterpreted, so natural numbers are interpreted as in the intended model, but ordinals are not:

$$
\begin{array}{rcl}
[\![\mathtt{N}]\!] & = & N \\
[\![\dot{\mathcal{O}}]\!] & = & \mathcal{O}_M
\end{array}
$$

where

$$\mathcal{O}_M : Set$$

has the following introduction rules:

$$\begin{array}{rcl} 0_M & : & \mathcal{O}_M \\ sup_M & : & (c : \mathtt{T}(\mathtt{N} \dot{\rightarrow} \dot{\mathcal{O}}); f : (N)\mathcal{O}_M)\mathcal{O}_M \end{array}$$

and the following recursion operator

$$ordrec_M : (C : Set; C; (\mathtt{T}(\mathtt{N} \dot{\rightarrow} \dot{\mathcal{O}}); (N)\mathcal{O}_M; (N)C)C; \mathcal{O}_M)C$$

The quote function has the following new clauses

$$\begin{array}{rcl} \mathbf{q}_{\mathtt{N}}(0) & = & \mathtt{0} \\ \mathbf{q}_{\mathtt{N}}(s(p)) & = & \mathtt{s}(\mathbf{q}_{\mathtt{N}}(p)) \\ \mathbf{q}_{\dot{\mathcal{O}}}(0_M) & = & \mathtt{0} \\ \mathbf{q}_{\dot{\mathcal{O}}}(sup_M(c,f)) & = & \mathtt{sup}(c) \end{array}$$

Term interpretation in the enriched model becomes

$$\begin{array}{rcl} [\![\mathtt{rec}(d,e)]\!] & = & \langle \mathtt{rec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])), \lambda([z]rec([\![d]\!], [x,y]app_M(app_M([\![e]\!],x),y),z))\rangle \\ [\![\mathtt{0}]\!] & = & 0_M \\ [\![\mathtt{sup}(b)]\!] & = & sup_M(\mathbf{q}([\![b]\!]), [x]app_M([\![b]\!],x)) \\ [\![\mathtt{ordrec}(d,e)]\!] & = & \langle \mathtt{ordrec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])), \\ & & \quad \lambda([z]ordrec_M([\![d]\!], \\ & & \qquad [a,xf,yf]app_M(app_M([\![e]\!], \langle a, \lambda(xf)\rangle), \langle \mathtt{ordrec}(\mathbf{q}([\![d]\!]), \mathbf{q}([\![e]\!])) \circ a, \lambda(yf)\rangle), \\ & & \qquad z))\rangle \end{array}$$

where we have omitted cases which are the same as for the intended interpretation.

3.3.4 Normalization proof

We extend the definition of reducible value:

$$\begin{array}{rcl} Gl_{\mathtt{N}}(p) & = & 1 \\ Gl_{\dot{\mathcal{O}}}(p) & = & Glord(p) \end{array}$$

where

$$Glord : (\mathcal{O}_M)Set$$

is inductively defined by the introduction rules

$$\begin{array}{l} Glord(0_M) \\ (c : \mathtt{T}(\mathtt{N} \dot{\rightarrow} \dot{\mathcal{O}}); f : (N)\mathcal{O}_M; (p : N)Glord(f(p)) \times (\mathtt{I}(\mathtt{app}(c, \mathbf{q}(p)), \mathbf{q}((f(p)))))) Glord(sup_M(c,f)) \end{array}$$

The normalization proof extends.

3.3.5 Proof that the constructors are one-to-one

We illustrate this point only in the case of natural numbers, though this method of proof works generally for any data type.

By definition of $\mathbf{q}()\mathtt{N}$, the constructor $\mathtt{s}$ commutes with the normalization function. Let us assume that $a, b : \mathtt{T}(\mathtt{N})$ satisfy $\mathtt{I}(\mathtt{s}(a), \mathtt{s}(b))$, we have then

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

since conversion implies identity of normal forms, and then

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

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

$$I(\mathbf{nf}(a), \mathbf{nf}(b))$$

because `s` is one-to-one for the *intensional* equality; from this follows

$$\mathtt{I}(a, b)$$

because any term is convertible to its normal form.

4 Typed λ-calculus

We shall consider normalization to weak head normal form, that is, no normalization under λ is performed. The method is similar to the treatment of combinatory logic, and we present the normalization algorithms but not the proofs. These are done in an analogous way.

Instead we focus on the choice of syntax for the typed λ-calculus. Shall we use traditional named variables, de Bruijn indices, or perhaps some kind of categorical combinators; shall we use a presentation à la Curry or à la Church; shall we employ *explicit* or *implicit* substitutions, etc? We shall make two basic choices. Firstly, as for combinatory logic we shall use formulations à la Church, which arise by first considering formalization of natural deduction systems for intuitionistic implicational calculus. This is one approach; we do not claim that it is the canonical one. (Compare for example Altenkirch [2] and Huet [14] who use ordinary de Bruijn-indices in their implementations of proofs about λ-calculi.) Secondly, we focus on normal terms, since these are the only ones we need for building models. One model can then be used for normalization proofs of several essentially equivalent calculi. We illustrate this by building a model which can be used for the normalization proofs of both a weak λ-calculus with explicit substitutions (similar to the $\lambda\sigma$-calculus [1]) and a nameless version of the weak λ-calculus with implicit substitutions which was used by Martin-Löf [20].

4.1 Variables

We begin by defining sets of variables. These will then be used for formalizing sets of normal terms and certain sets of general terms. We will also introduce variable sequences, which correspond to renamings. We shall in particular define the identity renaming.

The basic point is that the rule for assuming a variable in typed λ-calculus corresponds to the logical rule of assumption

$$(\Gamma \dot{\ni} A)\mathtt{T}(\Gamma, A)$$

stating that A is true iff it is a member of the assumption list (*context*) Γ. Usually, the membership requirement is stated as a side-condition but here it is an assumption. If we define membership inductively

$$\dot{\ni} : (\Gamma : \mathtt{Context}; A : \mathtt{Type})Set$$

$$\begin{array}{lcl} \mathtt{0} & : & (\Gamma : \mathtt{Context}; A : \mathtt{Type})\Gamma\#A\dot{\ni}A \\ \mathtt{s} & : & (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \Gamma\dot{\ni}A)\Gamma\#B\dot{\ni}A \end{array}$$

we get proof objects which correspond to bounded de Bruijn-indices, so we can think of $\Gamma\dot{\ni}A$ as the singleton set containing a variable of type A.

Contexts are defined by

$$\mathtt{Context} : Set$$

$$\begin{array}{rcl} \texttt{[]} & : & \texttt{Context} \\ . & : & (\texttt{Context};\texttt{Type})\texttt{Context} \end{array}$$

We also introduce variable lists:

$$\dot{\supseteq} : (\texttt{Context};\texttt{Context})Set$$

$$\begin{array}{rcl} \texttt{<>} & : & (\Delta : \texttt{Context})\Delta\dot{\supseteq}\texttt{[]} \\ \texttt{<,>} & : & (\Delta,\Gamma : \texttt{Context}; A : \texttt{Type}; \Gamma : \Delta\dot{\supseteq}\Gamma; \Delta\dot{\ni}A)\Delta\dot{\supseteq}\Gamma.A \end{array}$$

The notation is suggested by the fact that $\Delta\dot{\supseteq}\Gamma$ iff $\Gamma\dot{\ni}A$ implies $\Delta\dot{\ni}A$ for all A, that is, Γ is a subset of Δ if both are considered as *sets* of assumptions.

Just as lists of terms represent substitutions, lists of variables represent reindexing (nameless renaming). Logically, they represent structural manipulations of the context by weakening, contraction, and exchange. We will use the identity reindexing defined by

$$\texttt{id} : (\Gamma : \texttt{Context})\Gamma\dot{\supseteq}\Gamma$$

$$\begin{array}{rcl} \texttt{id}_{\texttt{[]}} & = & \texttt{<>} \\ \texttt{id}_{\Gamma.A} & = & \texttt{<s(id}_\Gamma\texttt{),0>} \end{array}$$

where

$$\texttt{s} : (\Delta,\Gamma : \texttt{Context}; B : \texttt{Type}; \Delta\dot{\supseteq}\Gamma)\Delta.B\dot{\supseteq}\Gamma$$

$$\begin{array}{rcl} \texttt{s(<>)} & = & \texttt{<>} \\ \texttt{s(<}vs, v\texttt{>)} & = & \texttt{<s(}vs\texttt{),s(}v\texttt{)>} \end{array}$$

is the weakening (lifting, projection) lifted to reindexings.

4.2 Normalization to weak head normal form

4.2.1 Normal terms

Weak head normal terms have either of the forms $f(a_1,\ldots,a_n)$ or $v(a_1,\ldots,a_n)$, where f is a λ-abstraction, v is a variable and $a_1,\ldots,a_n$ are weak head normal terms. Since f is a λ-abstraction it may contain an arbitrary (not necessary normal) term, and hence the definition of normal term `Nt` is parametric in the definition of general term `T`[8]. We define normal terms and lists of normal terms by a simultaneous inductive definition:

$$\begin{array}{rcl} \texttt{Nt} & : & (\texttt{Context};\texttt{Type})Set \\ \to_{\texttt{Nt}} & : & (\texttt{Context};\texttt{Context})Set \end{array}$$

$$\begin{array}{rcl} \texttt{appf} & : & (\Delta,\Gamma : \texttt{Context}; A,B : \texttt{Type}; \texttt{T}(\Gamma.A,B); \Delta \to_{\texttt{Nt}} \Gamma)\texttt{Nt}(\Delta, A\dot{\to}B) \\ \texttt{appv} & : & (\Delta,\Gamma : \texttt{Context}; A : \texttt{Type}; \Delta\dot{\ni}\Gamma\dot{\twoheadrightarrow}A; \Delta \to_{\texttt{Nt}} \Gamma)\texttt{Nt}(\Delta, A) \\ \texttt{<>} & : & (\Delta : \texttt{Context})\Delta \to_{\texttt{Nt}} \texttt{[]} \\ \texttt{<,>} & : & (\Delta,\Gamma : \texttt{Context}; A : \texttt{Type}; \Delta \to_{\texttt{Nt}} \Gamma; \texttt{Nt}(\Delta,A))\Delta \to_{\texttt{Nt}} \Gamma.A \end{array}$$

where $\dot{\twoheadrightarrow}$ is multivariate function space:

$$\dot{\twoheadrightarrow} : (\texttt{Context};\texttt{Type})\texttt{Type}$$

$$\begin{array}{rcl} (\Gamma.A)\dot{\twoheadrightarrow}B & = & \Gamma\dot{\twoheadrightarrow}(A\dot{\to}B) \\ \texttt{[]}\dot{\twoheadrightarrow}B & = & B \end{array}$$

[8]Alternatively, we could assume that λ-abstractions come normalized and replace `T` with `Nt` in the body of `appf`.

4.2.2 The glued model

We can now build our model. First we interpret types

$$[\![]\!] : (\texttt{Context}; \texttt{Type})Set$$

$$[\![A \dot{\to} B]\!]_\Gamma = \texttt{Nt}(\Gamma, A \dot{\to} B) \times ([\![A]\!]_\Gamma \to [\![B]\!]_\Gamma)$$

and the quote function

$$\mathbf{q} : (\Gamma : \texttt{Context}; A : \texttt{Type}; [\![A]\!]_\Gamma)\texttt{Nt}(\Gamma, A)$$

$$\mathbf{q}_{A \dot{\to} B}(\langle c, f\rangle) = c$$

This can be lifted to yield interpretation of contexts and quotation of lists of values as well. We overload notation for these:

$$[\![\ ]\!] : (\texttt{Context}; \texttt{Context})Set$$

$$\mathbf{q}(:)(\Delta, \Gamma : \texttt{Context}; [\![\Gamma]\!]_\Delta)\Delta \to_{\texttt{Nt}} \Gamma$$

We use three auxiliary functions. The first is the projection function

$$\pi : (\Delta, \Gamma : \texttt{Context}; A : \texttt{Type}; \Gamma \dot{\ni} A; [\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta$$

$$\begin{array}{rcl} \pi_{\texttt{0}}(\langle ps, p\rangle) & = & p \\ \pi_{\texttt{s}(v1)}(\langle ps, p\rangle) & = & \pi_{v1}(ps) \end{array}$$

The second is application in the model

$$app_M : (\Gamma : \texttt{Context}; A, B : \texttt{Type}; p : [\![A \dot{\to} B]\!]_\Gamma; q : [\![A]\!]_\Gamma)[\![B]\!]_\Gamma$$

$$app_M(\langle c, f\rangle, p) = app(f, p)$$

The third is identity in the model. We define it as the interpretation of the identity renaming. This is obtained by first interpreting variables and renamings in general:

$$[\![\ ]\!] : (\Delta, \Gamma : \texttt{Context}; A : \texttt{Type}; v : \Delta \dot{\ni} \Gamma \dot{\twoheadrightarrow} A; ps : [\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta$$

$$[\![v]\!]_{A \dot{\to} B}(ps) \quad = \quad \langle \texttt{appv}(v,' ps), \lambda([p][\![v]\!]_B(\langle ps, p\rangle))\rangle$$

This can be raised to an interpretation of lists of variables. Hence we get

$$id_M = [\![\texttt{id}]\!](\langle\rangle)$$

4.2.3 Interpretation of a calculus of substitutions

As an example we shall prove normalization to whnf for a calculus with explicit substitutions similar to the $\lambda\sigma$-calculus. The syntax of terms is the following

$$\begin{array}{rcl} \texttt{T} & : & (\Gamma : \texttt{Context}; A : \texttt{Type})Set \\ \longrightarrow & : & (\Delta, \Gamma : \texttt{Context})Set \end{array}$$

$$\begin{array}{rcl} \texttt{0} & : & (\Gamma : \texttt{Context}; A : \texttt{Type})\texttt{T}(\Gamma.A, A) \\ \texttt{s} & : & (\Gamma : \texttt{Context}; A, B : \texttt{Type}; \texttt{T}(\Gamma, B))\texttt{T}(\Gamma.A, B) \\ \texttt{app} & : & (\Gamma : \texttt{Context}; A, B : \texttt{Type}; \texttt{T}(\Gamma, A \dot{\to} B); \texttt{T}(\Gamma, A))\texttt{T}(\Gamma, B) \\ \dot{\lambda} & : & (\Gamma : \texttt{Context}; A, B : \texttt{Type}; \texttt{T}(\Gamma.A, B))\texttt{T}(\Gamma, A \dot{\to} B) \\ \texttt{[ ]} & : & (\Delta, \Gamma : \texttt{Context}; A : \texttt{Type}; \texttt{T}(\Gamma, A); \Delta \longrightarrow \Gamma)\texttt{T}(\Delta, A) \end{array}$$

$$\begin{array}{rcl} \texttt{id} & : & (\Gamma : \texttt{Context})\Gamma \longrightarrow \Gamma \\ \texttt{o} & : & (\Gamma, \Delta, \Theta : \texttt{Context}; \Delta \longrightarrow \Theta; \Gamma \longrightarrow \Delta)\Gamma \longrightarrow \Theta \\ \texttt{<>} & : & (\Gamma : \texttt{Context})\Gamma \longrightarrow \texttt{[]} \\ \texttt{<,>} & : & (\Delta, \Gamma : \texttt{Context}; A : \texttt{Type}; \Delta \longrightarrow \Gamma; \texttt{T}(\Delta, A))\Delta \longrightarrow \Gamma.A \end{array}$$

The interpretation of terms and term lists in the model is

$$\begin{array}{rl} [\![\ \ ]\!]: & (\Delta,\Gamma:\mathtt{Context};A:\mathtt{Type};\mathtt{T}(\Gamma,A);[\![\Gamma]\!]_\Delta)[\![A]\!]_\Delta \\ [\![\ \ ]\!]: & (\Theta,\Delta,\Gamma:\mathtt{Context};\Delta\longrightarrow\Gamma;[\![\Delta]\!]_\Theta)[\![\Gamma]\!]_\Theta \end{array}$$

$$\begin{array}{rcl} [\![\mathtt{0}]\!]\langle ps,p\rangle & = & p \\ [\![\mathtt{s}(a)]\!]\langle ps,p\rangle & = & [\![a]\!](ps) \\ [\![\mathtt{app}(c,a)]\!](ps) & = & app_M([\![c]\!](ps),[\![a]\!](ps)) \\ [\![\dot\lambda(b)]\!](ps) & = & \langle \mathtt{appf}(b,'ps),\lambda([p][\![b]\!]\langle ps,p\rangle)\rangle \\ [\![a\,\mathtt{[}as\mathtt{]}]\!](ps) & = & [\![a]\!]([\![as]\!](ps)) \end{array}$$

$$\begin{array}{rcl} [\![\mathtt{id}]\!](ps) & = & ps \\ [\![cs\mathtt{o}bs]\!](ps) & = & [\![cs]\!]([\![bs]\!](ps)) \\ [\![\mathtt{<>}]\!](ps) & = & \langle\rangle \\ [\![\mathtt{<}as,a\mathtt{>}]\!](ps) & = & \langle [\![as]\!](ps),[\![a]\!](ps)\rangle \end{array}$$

The normal form function is then

$$\mathbf{nf}(a)=\mathbf{q}([\![a]\!](id_M))$$

We have the following conversion rules. (Leaving out congruence rules)

$$\begin{array}{rcl} \mathtt{I}(\ ,\ ) & : & (\Gamma:\mathtt{Context};A:\mathtt{Type};\mathtt{T}(\Gamma,A);\mathtt{T}(\Gamma,A))Set \\ \mathtt{I}(\ ,\ ) & : & (\Delta,\Gamma:\mathtt{Context};\Delta\longrightarrow\Gamma;\Delta\longrightarrow\Gamma)Set \end{array}$$

$$\begin{array}{l} \mathtt{I}(\mathtt{app}(\dot\lambda(b)\,\mathtt{[}as\mathtt{]},a),b\,\mathtt{[<}as,a\mathtt{>]}) \\ \mathtt{I}(\mathtt{app}(c,a)\,\mathtt{[}as\mathtt{]},\mathtt{app}(c\,\mathtt{[}as\mathtt{]},c\,\mathtt{[}as\mathtt{]})) \\ \mathtt{I}(\mathtt{0}\,\mathtt{[<}as,a\mathtt{>]},a) \\ \mathtt{I}(\mathtt{s}(b)\,\mathtt{[<}as,a\mathtt{>]},b\,\mathtt{[}as\mathtt{]}) \\ \mathtt{I}(b\,\mathtt{[}bs\mathtt{]}\,\mathtt{[}as\mathtt{]},b\,\mathtt{[}bs\mathtt{o}as\mathtt{]}) \end{array}$$

$$\begin{array}{l} \mathtt{I}(\mathtt{<>o}as,\mathtt{<>}) \\ \mathtt{I}(\mathtt{<}bs,b\mathtt{>o}as,\mathtt{<}bs\mathtt{o}as,b\,\mathtt{[}as\mathtt{]>}) \\ \mathtt{I}(\mathtt{ido}as,as) \\ \mathtt{I}((cs\mathtt{o}bs)\mathtt{o}as,cs\mathtt{o}(bs\mathtt{o}as)) \end{array}$$

We got these rules when trying to prove properties of the semantics, compare the methodology with Knuth-Bendix completion used by Curien [9] for turning the equations for cartesian closed categories into a term rewriting system.

It now follows[9]

$$(\Gamma:\mathtt{Context};A:\mathtt{Type};a,a':\mathtt{T}(\Gamma,A);\mathtt{I}(a,a'))I([\![a]\!],[\![a']\!])$$

and hence

$$(\Gamma:\mathtt{Context};A:\mathtt{Type};a,a':\mathtt{T}(\Gamma,A);\mathtt{I}(a,a'))I(\mathbf{nf}(a),\mathbf{nf}(a'))$$

The normalization proof is to prove that

$$(\Gamma:\mathtt{Context};A:\mathtt{Type};a:\mathtt{T}(\Gamma,A))\mathtt{I}(a,\iota(\mathbf{nf}(a)))$$

[9]The mechanical proofs have not yet been completed

where

$$\iota : (\Gamma : \mathtt{Context}; A : \mathtt{Type}; \mathtt{Nt}(\Gamma, A))\mathtt{T}(\Gamma, A)$$

is the injection of normal terms into general terms.

The proof of this is similar to the proof for combinatory logic, but in addition to a notion of glued value, we also need a notion of glued list of values. We omit the details of this construction.

4.2.4 Interpretation of the calculus of Martin-Löf 1973

Can we treat normalization to whnf for the traditional syntax? Yes, but we must remember that we must also change the implicitely defined substitution function so that it does not go under λ (otherwise we lose confluence, see Curien, Hardin, and Lèvy [10]). Therefore Martin-Löf [20] used a calculus where λ is replaced by `appf`:

$$\begin{array}{rcl}
\mathtt{var} & : & (\Gamma : \mathtt{Context}; A : \mathtt{Type}; \Gamma \dot{\ni} A)\mathtt{T}(\Gamma, A) \\
\mathtt{appf} & : & (\Delta, \Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma.A, B); \Delta \longrightarrow \Gamma)\mathtt{T}(\Delta, A \dot{\to} B) \\
\mathtt{app} & : & (\Gamma : \mathtt{Context}; A, B : \mathtt{Type}; \mathtt{T}(\Gamma, A \dot{\to} B); \mathtt{T}(\Gamma, A))\mathtt{T}(\Gamma, B) \\
\\
\mathtt{<>} & : & (\Delta : \mathtt{Context})\Delta \longrightarrow \mathtt{[]} \\
\mathtt{<,>} & : & (\Delta, \Gamma : \mathtt{Context}; A : \mathtt{Type}; \Gamma : \Delta \longrightarrow \Gamma; \mathtt{T}(\Delta, A))\Delta \longrightarrow \Gamma.A
\end{array}$$

The interpretation functions for this calculus into the glued model can also easily be written (where clauses which are the same as for the calculus with explicit substitution above are omitted).

$$\begin{array}{rcl}
[\![\mathtt{var}(v)]\!](ps) & = & \pi_v(ps) \\
[\![\mathtt{appf}(b, as)]\!](ps) & = & \langle \mathtt{appf}(b, \mathbf{q}([\![as]\!](ps))), \lambda([p][\![b]\!](\langle [\![as]\!](ps), p\rangle))\rangle
\end{array}$$

The conversion rule is

$$\mathtt{I}(\mathtt{app}(\mathtt{appf}(b, as), a), b\mathtt{[<}as, a\mathtt{>]})$$

where `_[_]` and `_ o _` are defined by a simultaneous recursive definition:

$$\begin{array}{rcl}
\mathtt{var}(v)\mathtt{[}as\mathtt{]} & = & \pi_v(as) \\
\mathtt{appf}(b, bs)\mathtt{[}as\mathtt{]} & = & \mathtt{appf}(b, bs\mathtt{o}as) \\
\mathtt{app}(c, a)\mathtt{[}as\mathtt{]} & = & \mathtt{app}(c\mathtt{[}as\mathtt{]}, a\mathtt{[}as\mathtt{]}) \\
\\
\mathtt{<>o}as & = & \mathtt{<>} \\
\mathtt{<}bs, b\mathtt{>o}as & = & \mathtt{<}bs\mathtt{o}as, b\mathtt{[}as\mathtt{]>}
\end{array}$$

5 Related work

Catarina Coquand [6] has used a similar technique for full normalization with η-expansion in simply typed λ-calculus. She also proves a property of a normalization algorithm, which is composed by an interpretation function and a quote function. But we note the following essential differences

- A Kripke model is used instead of a glued model.
- Equality in the Kripke model is extensional, whereas it is sufficient to consider intensional equality between glued values.
- Her quote function is defined simultaneously with an auxiliary unquote function which is needed for interpreting variables. Both are defined by recursion on the type structure.

This algorithm is closely related to the normalization algorithm for full normalization with η-expansion in simply typed λ-calculus given by Berger and Schwichtenberg [5]. Their 'inversion of the evaluation functional' corresponds for example to the quote function from the Kripke model.

Categorical glueing was used by Lafont [15] for proving a coherence theorem for categorical combinators. However, he argued that the semantic component of his interpretation cannot directly be used for computing: 'Mais les *valeurs abstraites* de $A \rightarrow B$, avec leur composante fonctionelle, ne semblent guère "mechanisable" ' [15][page 18]. In contrast to this, we make use of the fact that these abstract values, when represented in our intuitionistic meta language, are indeed mechanizable. But, of course, the implementation of this meta language may still make use of an environment machine.

It is interesting to compare this situation with the following comments by Per Martin-Löf on a normalization result [18]: '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 In a sense, the process of 'understanding' is represented by the computation of the semantics of a term.

Similar algorithms have also been considered in the context of metacircularity and partial evaluation. Pfenning and Lee [25] considered a notion of metacircularity for the polymorphic λ-calculus and defined an 'approximately metacircular interpreter' similar to our 'intended semantics'. Mogensen [23] 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.

References

[1] M. Abadi, L. Cardelli, P.-L. Curien, and J.-J. Lévy. Explicit substitutions. In *ACM Conference on Principles of Programming Languages, San Francisco*, 1990.

[2] T. Altenkirch. A formalisation of the strong normalization proof for System F in LEGO. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications, Utrecht*, March 1993.

[3] H. P. Barendregt. Lambda calculi with types. In S. Abramsky, D. Gabbay, and T. Maibaum, editors, *Handbook of Logic in Computer Science*, volume 2, pages 118–310. Oxford University Press, 199?

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

[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. Analysis of simply typed lambda-calculus in ALF. In *Draft Proceedings of the Winter Meeting, Tanum Strand*, January 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 C. Paulin. Inductively defined types, preliminary version. In *LNCS 417, COLOG '88, International Conference on Computer Logic*. Springer-Verlag, 1990.

[9] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Pitman, 1986.

[10] P.-L. Curien, T. Hardin, and J.-J. Lévy. Confluence properties of weak and strong calculi of explicit substitution. Preliminary version, July 1991.

[11] P. Dybjer. Inductive families. *Formal Aspects of Computing.* To appear.

[12] 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.

[13] V. Gaspes and J. M. Smith. Machine checked normalization proofs for typed combinator calculi. In *Proceedings of the 1992 Workshop on Types for Proofs and Programs*, June 1992.

[14] G. Huet. Residual theory in λ-calculus: a complete Gallina development. 1993.

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

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

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

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

[19] 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.

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

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

[22] J. C. Mitchell and A. Scedrov. Notes on sconing and relators. 1992.

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

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

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

[26] Y. Takayama. Extraction of redundancy-free programs from constructive natural deduction proofs. Submitted for publication in Journal of Symbolic Computation.

[27] A. Tarski. Der wahrheitsbegriff in den formalisierten sprachen. *Studia Philosophica*, 1:261–405, 1936.

[28] J. W. Thatcher, E. G. Wagner, and J. B. Wright. More on advice on structuring compilers and proving them correct. *Theoretical Computer Science*, 15:223–249, 1981.