Primitive Recursion for Higher-Order Abstract Syntax

Joëlle Despeyroux[1], Frank Pfenning[2], and Carsten Schürmann[2]

[1] INRIA, F-06902 Sophia-Antipolis Cedex, France
joelle.despeyroux@sophia.inria.fr
[2] Carnegie Mellon University, Pittsburgh PA 15213, USA
{fp|carsten}@cs.cmu.edu

1 Introduction

Higher-order abstract syntax is a central representation technique in many logical frameworks, that is, meta-languages designed for the formalization of deductive systems. The basic idea is to represent variables of the object language by variables in the meta-language. Consequently, object language constructs which bind variables must be represented by meta-language constructs which bind the corresponding variables.

This deceptively simple idea, which goes back to Church [1] and Martin-Löf's system of arities [18], has far-reaching consequences for the methodology of logical frameworks. On one hand, encodings of logical systems using this idea are often extremely concise and elegant, since common concepts and operations such as variable binding, variable renaming, capture-avoiding substitution, or parametric and hypothetical judgments are directly supported by the framework and do not need to be encoded separately in each application. On the other hand, higher-order representations are no longer inductive in the usual sense, which means that standard techniques for reasoning by induction do not apply.

Various attempts have been made to preserve the advantages of higher-order abstract syntax in a setting with strong induction principles [5, 4], but none of these is entirely satisfactory from a practical or theoretical point of view.

In this paper we take a first step towards reconciling higher-order abstract syntax with induction by proposing a system of *primitive recursive functionals* that permits iteration over subjects of functional type. In order to avoid the well-known paradoxes which arise in this setting (see Section 3), we decompose the primitive recursive function space $A \Rightarrow B$ into a modal operator and a parametric function space $(\Box A) \to B$. The inspiration comes from linear logic which arises from a similar decomposition of the intuitionistic function space $A \supset B$ into a modal operator and a linear function space $(!A) \multimap B$.

The resulting system allows, for example, iteration over the structure of expressions from the untyped λ-calculus when represented using higher-order abstract syntax. It is general enough to permit iteration over objects of *any* simple type, constructed over *any* simply typed signature and thereby encompasses Gödel's system T [9]. Moreover, it is conservative over the simply-typed λ-calculus which means that the compositional adequacy of encodings in higher-order abstract syntax is preserved. We view our calculus as an important first step towards a system which allows the methodology of logical frameworks such as LF [10] to be incorporated into systems such as Coq [20] or ALF [12].

The remainder of this paper is organized as follows: Section 2 reviews the idea of higher order abstract syntax and introduces the simply typed λ-calculus ($\lambda^{\rightarrow}$) which we extend to a modal λ-calculus in Section 3. Section 4 then presents the concept of iteration. In Section 5 we sketch the proof of our central result, namely that our extension is conservative over $\lambda^{\rightarrow}$. Finally, Section 6 assesses the results, compares some related work, and outlines future work. A full version of this paper with complete technical developments and detailed proofs is accessible as `http://www.cs.cmu.edu/~carsten/CMU-CS-96-172.ps.gz` [6].

2 Higher-Order Abstract Syntax

Higher-order abstract syntax exploits the full expressive power of a typed λ-calculus for the representation of an object language, where λ-abstraction provides the mechanism to represent binding. In this paper, we restrict ourselves to a simply typed meta-language, although we recognize that an extension allowing dependent types and polymorphism is important future work (see Section 6). Our formulation of the simply-typed meta-language is standard.

$$\begin{array}{lll} \text{Pure types:} & B & ::= a \mid B_1 \rightarrow B_2 \\ \text{Objects:} & M & ::= x \mid c \mid \lambda x\!:\!A.\, M \mid M_1\ M_2 \\[1ex] \text{Context:} & \Psi & ::= \cdot \mid \Psi, x : B \\ \text{Signature:} & \Sigma & ::= \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c : B \end{array}$$

We use a for type constants, c for object constants and x for variables. We also fix a signature Σ for our typing and evaluation judgments so we do not have to carry it around.

Definition 1 Typing judgment. $\Psi \vdash M : B$ is defined by:

$$\frac{\Psi(x) = B}{\Psi \vdash x : B}\,\mathsf{StpVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c : B}\,\mathsf{StpConst}$$

$$\frac{\Psi, x : B_1 \vdash M : B_2}{\Psi \vdash \lambda x\!:\!B_1.\, M : B_1 \rightarrow B_2}\,\mathsf{StpLam} \qquad \frac{\Psi \vdash M_1 : B_2 \rightarrow B_1 \qquad \Psi \vdash M_2 : B_2}{\Psi \vdash M_1\ M_2 : B_1}\,\mathsf{StpApp}$$

As running examples throughout the paper we use the representation of natural numbers and untyped λ-expressions.

Example 1 Natural numbers.

$$\begin{array}{rcl} & & \mathrm{nat} : \mathrm{type} \\ \ulcorner 0 \urcorner = \mathrm{z} & & \mathrm{z} \quad : \mathrm{nat} \\ \ulcorner n+1 \urcorner = \mathrm{s}\ \ulcorner n \urcorner & & \mathrm{s} \quad : \mathrm{nat} \rightarrow \mathrm{nat} \end{array}$$

Untyped λ-expressions illustrate the idea of higher-order abstract syntax: object language variables are represented by meta-language variables.

Example 2 Untyped λ-expressions.

$$\text{Expressions: } e ::= x \mid \mathbf{lam}\ x.e \mid e_1 @ e_2$$

$$\begin{array}{rcl@{\qquad}l}
 & & & \mathrm{exp} : \mathrm{type} \\
\ulcorner \mathbf{lam}\ x.e \urcorner & = & \mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, \ulcorner e \urcorner) & \mathrm{lam} : (\mathrm{exp} \to \mathrm{exp}) \to \mathrm{exp} \\
\ulcorner e_1 @ e_2 \urcorner & = & \mathrm{app}\ \ulcorner e_1 \urcorner\, \ulcorner e_2 \urcorner & \mathrm{app} : \mathrm{exp} \to (\mathrm{exp} \to \mathrm{exp}) \\
\ulcorner x \urcorner & = & x &
\end{array}$$

Not every well-typed object of the meta-language directly represents an expression of the object language. For example, we can see that $\ulcorner e \urcorner$ will never contain a β-redex. Moreover, the argument to lam which has type exp $\to$ exp will always be a λ-abstraction. Thus the image of the translation in this representation methodology is always a β-normal and η-long form. Following [10], we call these forms *canonical* as defined by the following two judgments.

Definition 2 Atomic and canonical forms.

1. $\Psi \vdash V \downarrow B$ (V is atomic of type B in Ψ)
2. $\Psi \vdash V \Uparrow B$ (V is canonical of type B in Ψ)

are defined by:

$$\frac{\Psi(x) = B}{\Psi \vdash x \downarrow B}\,\mathsf{AtVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c \downarrow B}\,\mathsf{AtConst} \qquad \frac{\Psi \vdash V_1 \downarrow B_2 \to B_1 \qquad \Psi \vdash V_2 \Uparrow B_2}{\Psi \vdash V_1\ V_2 \downarrow B_1}\,\mathsf{AtApp}$$

$$\frac{\Psi \vdash V \downarrow a}{\Psi \vdash V \Uparrow a}\,\mathsf{CanAt} \qquad \frac{\Psi, x : B_1 \vdash V \Uparrow B_2}{\Psi \vdash \lambda x\!:\!B_1.\, V \Uparrow B_1 \to B_2}\,\mathsf{CanLam}$$

Canonical forms play the role of "observable values" in a functional language: they are in one-to-one correspondence with the expressions we are trying to represent. For Example 2 (untyped λ-expressions) this is expressed by the following property, which is proved by simple inductions.

Example 3 Compositional adequacy for untyped λ-expressions.

1. Let e be an expression with free variables among $x_1, \ldots, x_n$. Then $x_1 : \mathrm{exp}, \ldots, x_n : \mathrm{exp} \vdash \ulcorner e \urcorner \Uparrow \mathrm{exp}$.
2. Let $x_1 : \mathrm{exp}, \ldots, x_n : \mathrm{exp} \vdash M \Uparrow \mathrm{exp}$. Then $M = \ulcorner e \urcorner$ for an expression e with free variables among $x_1, \ldots, x_n$.
3. $\ulcorner \cdot \urcorner$ is a bijection between expressions and canonical forms where $\ulcorner [e'/x]e \urcorner = [\ulcorner e' \urcorner / x] \ulcorner e \urcorner$.

Since every object in $\lambda^{\to}$ has a unique $\beta\eta$-equivalent canonical form, the meaning of every well-typed object is unambiguously given by its canonical form. Our operational semantics (see Definitions 4 and 7) computes this canonical form and therefore the meaning of every well-typed object. That this property is preserved under an extension of the language by primitive recursion for higher-order abstract syntax may be considered the main technical result of this paper.

3 Modal λ-Calculus

The constructors for objects of type exp from Example 2 are lam : $(\text{exp} \to \text{exp}) \to \text{exp}$ and app : $\text{exp} \to (\text{exp} \to \text{exp})$. These cannot be the constructors of an *inductive* type exp, since we have a negative occurrence of exp in the argument type of lam. This is not just a formal observation, but has practical consequences: we cannot formulate a consistent induction principle for expressions in this representation. Furthermore, if we increase the computational power of the meta-language by adding **case** or an iterator, then not every well-typed object of type exp has a canonical form. For example,

$$\cdot \vdash \text{lam}\ (\lambda E\!:\!\text{exp}.\,\text{case}\ E\ \text{of app}\ E_1\ E_2 \Rightarrow \text{app}\ E_2\ E_1 \mid \text{lam}\ E' \Rightarrow \text{lam}\ E') : \text{exp}$$

but the given object does not represent any untyped λ-expression, nor could it be converted to one. The difficulty with a **case** or iteration construct is that there are many new functions of type $\text{exp} \to \text{exp}$ which cannot be converted to a function in $\lambda^{\to}$. This becomes a problem when such functions are arguments to constructors, since then the extension is no longer conservative even over expressions of base type (as illustrated in the example above).

Thus we must cleanly separate the *parametric function space* $\text{exp} \to \text{exp}$ whose elements are convertible to the form $\lambda x : \text{exp}.\, E$ where E is built only from the constructors app, lam, and the variable x, from the *primitive recursive function space* $\text{exp} \Rightarrow \text{exp}$ which is intended to encompass functions defined through case distinction and iteration. This separation can be achieved by using a modal operator: $\text{exp} \to \text{exp}$ will continue to contain only the parametric functions, while $\text{exp} \Rightarrow \text{exp} = (\Box\text{exp}) \to \text{exp}$ contains the primitive recursive functions.

Intuitively, we interpret $\Box B$ as the type of *closed* objects of type B. We can iterate or distinguish cases over closed objects, since all constructors are statically known and can be provided for. This is not the case if an object may contain some unknown free variables. The system is non-trivial since we may also abstract over objects of type $\Box A$, but fortunately it is well understood and corresponds (via an extension of the Curry-Howard isomorphism) to the intuitionistic variant of S_4 [3].

In Section 4 we introduce schemas for defining functions by iteration and case distinction which require the subject to be of type $\Box B$. We can recover the ordinary scheme of primitive recursion for type nat if we also add pairs to the language. Pairs (with type $A_1 \times A_2$) are also necessary for the simultaneous definition of mutually recursive functions. Just as the modal type $\Box A$, pairs are lazy and values of these types are not observable—ultimately we are only interested in canonical forms of pure type.

The formulation of the modal λ-calculus below is copied from [3] and goes back to [22]. The language of types includes the pure types from the simply-typed λ-calculus in Section 2.

$$\frac{\Gamma(x) = A}{\Delta; \Gamma \vdash x : A}\ \mathsf{TpVarReg} \qquad \frac{\Delta(x) = A}{\Delta; \Gamma \vdash x : A}\ \mathsf{TpVarMod} \qquad \frac{\Sigma(c) = B}{\Delta; \Gamma \vdash c : B}\ \mathsf{TpConst}$$

$$\frac{\Delta; \Gamma, x : A_1 \vdash M : A_2}{\Delta; \Gamma \vdash \lambda x{:}A_1.\, M : A_1 \to A_2}\ \mathsf{TpLam}$$

$$\frac{\Delta; \Gamma \vdash M_1 : A_2 \to A_1 \qquad \Delta; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash M_1\ M_2 : A_1}\ \mathsf{TpApp}$$

$$\frac{\Delta; \Gamma \vdash M_1 : A_1 \qquad \Delta; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash \langle M_1, M_2 \rangle : A_1 \times A_2}\ \mathsf{TpPair}$$

$$\frac{\Delta; \Gamma \vdash M : A_1 \times A_2}{\Delta; \Gamma \vdash \mathrm{fst}\ M : A_1}\ \mathsf{TpFst} \qquad \frac{\Delta; \Gamma \vdash M : A_1 \times A_2}{\Delta; \Gamma \vdash \mathrm{snd}\ M : A_2}\ \mathsf{TpSnd}$$

$$\frac{\Delta; \cdot \vdash M : A}{\Delta; \Gamma \vdash \mathrm{box}\ M : \Box A}\ \mathsf{TpBox} \qquad \frac{\Delta; \Gamma \vdash M_1 : \Box A_1 \qquad \Delta, x : A_1; \Gamma \vdash M_2 : A_2}{\Delta; \Gamma \vdash \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 : A_2}\ \mathsf{TpLet}$$

Fig. 1. Typing judgment $\Delta; \Gamma \vdash M : A$

$$\begin{array}{lll}
\text{Types:} & A & ::= a \mid A_1 \to A_2 \mid \Box A \mid A_1 \times A_2 \\
\text{Objects:} & M & ::= c \mid x \mid \lambda x{:}A.\, M \mid M_1\ M_2 \\
& & \quad \mid \mathrm{box}\ M \mid \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 \mid \langle M_1, M_2 \rangle \mid \mathrm{fst}\ M \mid \mathrm{snd}\ M \\
\text{Contexts:} & \Gamma & ::= \cdot \mid \Gamma, x : A
\end{array}$$

For the sake of brevity we usually suppress the fixed signature Σ. However, it is important that signatures Σ and contexts denoted by Ψ will continue to contain only pure types, while contexts Γ and Δ may contain arbitrary types. We also continue to use B to range over pure types, while A ranges over arbitrary types. The typing judgment $\Delta; \Gamma \vdash M : A$ uses two contexts: Δ, whose variables range over closed objects, and Γ, whose variables range over arbitrary objects.

Definition 3 Typing judgment. $\Delta; \Gamma \vdash M : A$ is defined in Figure 1.

As examples, we show some basic laws of the (intuitionistic) modal logic S_4.

Example 4 Laws of S_4.

$$\begin{array}{ll}
\mathrm{funlift} & : \Box(A_1 \to A_2) \to \Box A_1 \to \Box A_2 \\
& = \lambda f{:}\Box(A_1 \to A_2).\, \lambda x{:}\Box A_1. \\
& \qquad \mathrm{let\ box}\ f' = f\ \mathrm{in\ let\ box}\ x' = x\ \mathrm{in\ box}\ (f'\ x') \\[4pt]
\mathrm{unbox} & : \Box A \to A \\
& = \lambda x{:}\Box A.\, \mathrm{let\ box}\ x' = x\ \mathrm{in}\ x' \\[4pt]
\mathrm{boxbox} & : \Box A \to \Box\Box A \\
& = \lambda x{:}\Box A.\, \mathrm{let\ box}\ x' = x\ \mathrm{in\ box}\ (\mathrm{box}\ x')
\end{array}$$

$$\frac{\Psi \vdash M \hookrightarrow V : a}{\Psi \vdash M \Uparrow V : a}\,\textsf{EcAtomic} \qquad \frac{\Psi, x : B_1 \vdash M\ x \Uparrow V : B_2}{\Psi \vdash M \Uparrow \lambda x{:}B_1.\, V : B_1 \to B_2}\,\textsf{EcArrow}$$

$$\frac{\Psi(x) = A}{\Psi \vdash x \hookrightarrow x : A}\,\textsf{EvVar} \qquad \frac{\Sigma(c) = B}{\Psi \vdash c \hookrightarrow c : B}\,\textsf{EvConst}$$

$$\frac{\cdot; \Psi, x : A_1 \vdash M : A_2}{\Psi \vdash \lambda x{:}A_1.\, M \hookrightarrow \lambda x{:}A_1.\, M : A_1 \to A_2}\,\textsf{EvLam}$$

$$\frac{\begin{array}{l}\Psi \vdash M_1 \hookrightarrow \lambda x{:}A_2.\, M_1' : A_2 \to A_1 \\ \Psi \vdash M_2 \hookrightarrow V_2 : A_2 \\ \Psi \vdash [V_2/x](M_1') \hookrightarrow V : A_1\end{array}}{\Psi \vdash M_1\ M_2 \hookrightarrow V : A_1}\,\textsf{EvApp} \qquad \frac{\begin{array}{l}\Psi \vdash M_1 \hookrightarrow V_1 : B_2 \to B_1 \\ \Psi \vdash V_1 \downarrow B_2 \to B_1 \\ \Psi \vdash M_2 \Uparrow V_2 : B_2\end{array}}{\Psi \vdash M_1\ M_2 \hookrightarrow V_1\ V_2 : B_1}\,\textsf{EvAtomic}$$

$$\frac{\cdot; \Psi \vdash M_1 : A_1 \qquad \cdot; \Psi \vdash M_2 : A_2}{\Psi \vdash \langle M_1, M_2 \rangle \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2}\,\textsf{EvPair}$$

$$\frac{\Psi \vdash M \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \qquad \Psi \vdash M_1 \hookrightarrow V : A_1}{\Psi \vdash \mathrm{fst}\ M \hookrightarrow V : A_1}\,\textsf{EvFst}$$

$$\frac{\Psi \vdash M \hookrightarrow \langle M_1, M_2 \rangle : A_1 \times A_2 \qquad \Psi \vdash M_2 \hookrightarrow V : A_2}{\Psi \vdash \mathrm{snd}\ M \hookrightarrow V : A_2}\,\textsf{EvSnd}$$

$$\frac{\cdot; \cdot \vdash M : A}{\Psi \vdash \mathrm{box}\ M \hookrightarrow \mathrm{box}\ M : \Box A}\,\textsf{EvBox}$$

$$\frac{\Psi \vdash M_1 \hookrightarrow \mathrm{box}\ M_1' : \Box A \qquad \Psi \vdash [M_1'/x](M_2) \hookrightarrow V : A_2}{\Psi \vdash \mathrm{let\ box}\ x = M_1\ \mathrm{in}\ M_2 \hookrightarrow V : A_2}\,\textsf{EvLet}$$

Fig. 2. Evaluation judgments $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$

The rules for evaluation must be constructed in such a way that full canonical forms are computed for objects of pure type, that is, we must evaluate under certain λ-abstractions. Objects of type $\Box A$ or $A_1 \times A_2$ on the other hand are not observable and may be computed lazily. We therefore use two mutually recursive judgments for evaluation and conversion to canonical form, written $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$, respectively. The latter is restricted to pure types, since only objects of pure type possess canonical forms. Since we evaluate under some λ-abstractions, free variables of pure type declared in Ψ may occur in M and V during evaluation.

Definition 4 Evaluation judgment. $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$ are defined in Figure 2.

4 Iteration

The modal operator $\Box$ introduced in Section 3 allows us to restrict iteration and case distinction to subjects of type $\Box B$, where B is a pure type. The technical realization of this idea in its full generality is rather complex. We therefore begin by describing the behavior of functions defined by iteration informally, incrementally developing their formal definition within our system. In the informal presentation we elide the box constructor, but we should convince ourselves that the subject of the iteration or case is indeed assumed to be closed.

Example 5 Addition. The usual type of addition is $\mathrm{nat} \rightarrow \mathrm{nat} \rightarrow \mathrm{nat}$. This is no longer a valid type for addition, since it must iterate over either its first or second argument and would therefore not be parametric in that argument. Among the possible types for addition, we will be interested particularly in $\Box\mathrm{nat} \rightarrow \mathrm{nat} \rightarrow \mathrm{nat}$ and $\Box\mathrm{nat} \rightarrow \Box\mathrm{nat} \rightarrow \Box\mathrm{nat}$.

$$\begin{aligned}
&\mathrm{plus}\ \mathrm{z}\ n &&= n \\
&\mathrm{plus}\ (\mathrm{s}\ m)\ n &&= \mathrm{s}\ (\mathrm{plus}\ m\ n)
\end{aligned}$$

Note that this definition cannot be assigned type $\mathrm{nat} \rightarrow \mathrm{nat} \rightarrow \mathrm{nat}$ or $\Box\mathrm{nat} \rightarrow \mathrm{nat} \rightarrow \Box\mathrm{nat}$.

In our system we view iteration as replacing constructors of a canonical term by functions of appropriate type, which is also the idea behind *catamorphisms* [8]. In the case of natural numbers, we replace $\mathrm{z} : \mathrm{nat}$ by a term $M_z : A$ and $\mathrm{s} : \mathrm{nat} \rightarrow \mathrm{nat}$ by a function $M_s : A \rightarrow A$. Thus iteration over natural numbers replaces type nat by A. We use the notation $a \mapsto A$ for a *type replacement* and $c \mapsto M$ for a *term replacement.* Iteration in its simplest form is written as "it $\langle a \mapsto A\rangle\ M\ \langle \Omega\rangle$" where M is the subject of the iteration, and Ω is a list containing term replacements for all constructors of type a. The formal typing rules for replacements are given later in this section; first some examples.

Example 6 Addition via iteration. Addition from Example 5 can be formulated in a number of ways with an explicit iteration operator. The simplest one:

$$\begin{aligned}
\mathrm{plus}' \ &: \ \Box\mathrm{nat} \rightarrow \mathrm{nat} \rightarrow \mathrm{nat} \\
&= \lambda m\!:\!\Box\mathrm{nat}.\, \lambda n\!:\!\mathrm{nat}.\, \mathrm{it}\ \langle \mathrm{nat} \mapsto \mathrm{nat}\rangle\ m\ \langle \mathrm{z} \mapsto n \mid \mathrm{s} \mapsto \mathrm{s}\rangle
\end{aligned}$$

Later examples require addition with a result guaranteed to be closed. Its definition is only slightly more complicated.

$$\begin{aligned}
\mathrm{plus} \ &: \ \Box\mathrm{nat} \rightarrow \Box\mathrm{nat} \rightarrow \Box\mathrm{nat} \\
&= \lambda m\!:\!\Box\mathrm{nat}.\, \lambda n\!:\!\Box\mathrm{nat}.\, \mathrm{it}\ \langle \mathrm{nat} \mapsto \Box\mathrm{nat}\rangle\ m \\
&\qquad \langle\, \mathrm{z} \mapsto n \\
&\qquad \mid \mathrm{s} \mapsto (\lambda r\!:\!\Box\mathrm{nat}.\, \mathrm{let\ box}\ r' = r\ \mathrm{in\ box}\ (\mathrm{s}\ r'))\rangle
\end{aligned}$$

If the data type is higher-order, iteration over closed objects must traverse terms with free variables. We model this in the informal presentation by introducing new parameters (written as $\nu x\!:\!B.\, M$) using Odersky's notation [19]. This makes a dynamic extension of the function definition necessary to encompass the new parameters (written as "where $f(x) = M$").

Example 7 Counting variable occurrences. Below is a function which counts the number of occurrences of bound variables in an untyped λ-expression in the representation of Example 2. It can be assigned type $\Box\mathrm{exp} \to \Box\mathrm{nat}$.

$$\begin{array}{ll} \mathrm{cntvar}\ (\mathrm{app}\ e_1\ e_2) & = \mathrm{plus}\ (\mathrm{cntvar}\ e_1)\ (\mathrm{cntvar}\ e_2) \\ \mathrm{cntvar}\ (\mathrm{lam}\ e) & = \nu x\!:\!\mathrm{exp}.\, \mathrm{cntvar}\ (e\ x)\ \mathrm{where}\ \mathrm{cntvar}\ x = (\mathrm{s}\ \mathrm{z}) \end{array}$$

It may look like the recursive call in the example above is not well-typed since $(e\ x)$ is not closed as required, but contains a free parameter x. Making sense of this apparent contradiction is the principal difficulty in designing an iteration construct for higher-order abstract syntax. As before, we model iteration via replacements. Here, $\mathrm{exp} \mapsto \Box\mathrm{nat}$ and so $\mathrm{lam} \mapsto M_1$ and $\mathrm{app} \mapsto M_2$ where $M_1 : (\Box\mathrm{nat} \to \Box\mathrm{nat}) \to \Box\mathrm{nat}$ and $M_2 : \Box\mathrm{nat} \to (\Box\mathrm{nat} \to \Box\mathrm{nat})$. The types of replacement terms M_1 and M_2 arise from the types of the constructors $\mathrm{lam} : (\mathrm{exp} \to \mathrm{exp}) \to \mathrm{exp}$ and $\mathrm{app} : \mathrm{exp} \to (\mathrm{exp} \to \mathrm{exp})$ by applying the type replacement $\mathrm{exp} \mapsto \Box\mathrm{nat}$. We write

$$\begin{array}{ll} \mathrm{cntvar} & : \Box\mathrm{exp} \to \Box\mathrm{nat} \\ & = \lambda x\!:\!\Box\mathrm{exp}.\, \mathrm{it}\ \langle \mathrm{exp} \mapsto \Box\mathrm{nat}\rangle\ x \\ & \qquad \langle\ \mathrm{app} \mapsto \mathrm{plus} \\ & \qquad \mid \mathrm{lam} \mapsto \lambda f\!:\!\Box\mathrm{nat} \to \Box\mathrm{nat}.\, f\ (\mathrm{box}\ (\mathrm{s}\ \mathrm{z}))\rangle \end{array}$$

Informally, the result of $\mathrm{cntvar}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x))$ can be determined as follows:

$$\begin{array}{l} \mathrm{cntvar}\ (\mathrm{lam}\ (\lambda x\!:\!\mathrm{exp}.\, x)) \\ = \nu x'\!:\!\mathrm{exp}.\, \mathrm{cntvar}\ ((\lambda x\!:\!\mathrm{exp}.\, x)\ x')\ \mathrm{where}\ \mathrm{cntvar}\ x' = (\mathrm{s}\ \mathrm{z}) \\ = \nu x'\!:\!\mathrm{exp}.\, \mathrm{cntvar}\ x'\ \mathrm{where}\ \mathrm{cntvar}\ x' = (\mathrm{s}\ \mathrm{z}) \\ = \nu x'\!:\!\mathrm{exp}.\, (\mathrm{s}\ \mathrm{z})\ \mathrm{where}\ \mathrm{cntvar}\ x' = (\mathrm{s}\ \mathrm{z}) \\ = (\mathrm{s}\ \mathrm{z}) \end{array}$$

For the formal operational semantics, see Example 10.

A number of functions can be defined elegantly in this representation. Among them are the conversion from type exp to a representation using de Bruijn indices and one-step parallel reduction. The latter requires mutual iteration and pairs (see [6]).

The following example illustrates two concepts: mutually inductive types and iteration over the form of a (parametric!) function.

Example 8 Substitution in normal forms. Substitution is already directly definable by application, but one may also ask if there is a structural definition in the style of [16]. Normal forms of the untyped λ-calculus are represented by the type nf with an auxiliary definition for atomic forms of type at.

$$\begin{array}{l} \text{Normal forms: } N ::= P \mid \mathbf{lam}\ x.N \\ \text{Atomic forms: } P ::= x \mid P@N \end{array}$$

In this example the represention function $\ulcorner . \urcorner$ acts on normal forms, atomic forms are represented by $\ulcorner\!\ulcorner . \urcorner\!\urcorner$.

$$
\begin{array}{rcl}
 & & \mathrm{nf} \quad : \mathrm{type} \\
 & & \mathrm{at} \quad : \mathrm{type} \\
\ulcorner P \urcorner = \mathrm{atnf}\, \ulcorner\!\ulcorner P \urcorner\!\urcorner & & \mathrm{atnf} : \mathrm{at} \to \mathrm{nf} \\
\ulcorner \mathbf{lam}\ x.N \urcorner = \mathrm{lm}\ (\lambda x\!:\!\mathrm{at}.\ulcorner N \urcorner) & & \mathrm{lm} \quad : (\mathrm{at} \to \mathrm{nf}) \to \mathrm{nf} \\
\ulcorner\!\ulcorner P @ N \urcorner\!\urcorner = \mathrm{ap}\ \ulcorner\!\ulcorner P \urcorner\!\urcorner\, \ulcorner N \urcorner & & \mathrm{ap} \quad : \mathrm{at} \to \mathrm{nf} \to \mathrm{at} \\
\ulcorner\!\ulcorner x \urcorner\!\urcorner = x & &
\end{array}
$$

Substitution of atomic objects for variables is defined by two mutually recursive functions, one with type $\mathrm{subnf} : \Box(\mathrm{at} \to \mathrm{nf}) \to \mathrm{at} \to \mathrm{nf}$ and $\mathrm{subat} : \Box(\mathrm{at} \to \mathrm{at}) \to \mathrm{at} \to \mathrm{at}$.

$$
\begin{array}{ll}
\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\,\mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\,N\ x\ y))\ Q & = \mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\,\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\,(N\ x\ y))\ Q \\
 & \qquad \text{where } \mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\,y)\ Q = y) \\
\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\,\mathrm{atnf}\ (P\ x))\ Q & = \mathrm{atnf}\ (\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\,P\ x)\ Q) \\
\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\,\mathrm{ap}\ (P\ x)\ (N\ x))\ Q & = \mathrm{ap}\ (\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\,P\ x)\ Q) \\
 & \qquad (\mathrm{subnf}\ (\lambda x\!:\!\mathrm{at}.\,N\ x)\ Q) \\
\mathrm{subat}\ (\lambda x\!:\!\mathrm{at}.\,x)\ Q & = Q
\end{array}
$$

The last case arises since the parameter x must be considered as a new constructor in the body of the abstraction. The functions above are realized in our calculus by a simultaneous replacement of objects of type nf and at. In other words, the type replacement must account for all mutually recursive types, and the term replacement for all constructors of those types.

$$
\begin{array}{l}
\mathrm{subnf}\ :\ \Box(\mathrm{at} \to \mathrm{nf}) \to \mathrm{at} \to \mathrm{nf} \\
\quad = \lambda N\!:\!\Box(\mathrm{at} \to \mathrm{nf}).\,\lambda Q\!:\!\mathrm{at}.\,\mathrm{it}\ \langle \mathrm{nf} \mapsto \mathrm{nf} \mid \mathrm{at} \mapsto \mathrm{at} \rangle\ N \\
\qquad\qquad \langle\ \mathrm{lm} \mapsto \lambda F\!:\!\mathrm{at} \to \mathrm{nf}.\,\mathrm{lm}\ (\lambda y\!:\!\mathrm{at}.\,(F\ y)) \\
\qquad\qquad \mid \mathrm{atnf} \mapsto \lambda P\!:\!\mathrm{at}.\,\mathrm{atnf}\ P \\
\qquad\qquad \mid \mathrm{ap} \mapsto \lambda P\!:\!\mathrm{at}.\,\lambda N\!:\!\mathrm{nf}.\,\mathrm{ap}\ P\ N \rangle \\
\qquad\qquad Q
\end{array}
$$

Via η-contraction we can see that substitution amounts to a structural identity function.

We begin now with the formal discussion and description of the full language. Due to the possibility of mutual recursion among types, the type replacements must be lists (see Example 8).

$$\text{Type replacement: } \omega ::= \cdot \mid (\omega \mid a \mapsto A)$$

Which types must be replaced by an iteration depends on which types are mutually recursive according to the constructors in the signature Σ and possibly the type of the iteration subject itself. If we iterate over a function, the parameter of a function must be treated like a constructor for its type, since it can appear in that role in the body of a function.

Thus, we define the notion of *type subordination* which summarizes all dependencies between atomic types by separately considering its *static* part $\lhd_\Sigma$ which derives from the dependencies induced by the constructor types from Σ and its *dynamic* part $\lhd_B$ which accounts for dependencies induced from the argument types of B. The transitive closure $\blacktriangleleft_{\Sigma;B}$ of static and dynamic type subordination relation defines cleanly all dependencies between types which govern the formation of the subject of iteration. We denote the *target type* of a pure type B by $\tau(B)$. All type constants which are mutually dependent with $\tau(B)$, written $\mathcal{I}(\Sigma; B)$, form the domain of the type replacement ω:

$$\mathcal{I}(\Sigma; B) := \{a | \tau(B) \blacktriangleleft_{\Sigma;B} a \text{ and } a \blacktriangleleft_{\Sigma;B} \tau(B)\}$$

In Example 8 of normal and atomic forms we have $\mathcal{I}(\Sigma; \mathrm{at} \to \mathrm{nf}) = \{\mathrm{at}, \mathrm{nf}\}$. Note that type subordination is built into calculi where inductive types are defined explicitly (such as the Calculus of Inductive Constructions [20]); here it must be recovered from the signature since we impose no ordering constraints except that a type must be declared before it is used. Our choice to recover the type subordination relation from the signature allows us to perform iteration over any functional type, without fixing the possibilities in advance.

Let us now address the question of how the type of an iteration is formed: If the subject of iteration has type B, the iterator object has type $\langle\omega\rangle(B)$, where $\langle\omega\rangle(B)$ is defined inductively by replacing each type constant according to ω, leaving types outside the domain fixed.

A similar replacement is applied at the level of terms: the result of an iteration is an object which resembles the (canonical) subject of the iteration in structure, but object constants are replaced by other objects carrying the intended computational meaning of the different cases. Even though the subject of iteration is closed at the beginning of the replacement process, we need to deal with embedded λ-abstractions due to higher-order abstract syntax. But since such functions are parametric we can simply replace variables x of type B by new variables x' of type $\langle\omega\rangle(B)$.

$$\text{Term replacement: } \Omega ::= \cdot \mid (\Omega \mid c \mapsto M) \mid (\Omega \mid x \mapsto x')$$

Initially the domain of a term replacement is a signature containing all constructors whose target type is in $\mathcal{I}(\Sigma; B)$. We refer to this signature as $\mathcal{S}_{\mathrm{it}}(\Sigma; B)$. The form of iteration follows now quite naturally: We extend the notion of objects by

$$M ::= \ldots \mid \mathrm{it}\ \langle\omega\rangle\ M\ \langle\Omega\rangle$$

and define the following typing rules for iteration and term replacements.

Definition 5 Typing judgment for iteration. (extending Definition 3)

$$\frac{\Delta; \Gamma \vdash M : \Box B \qquad \Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\mathcal{S}_{\mathrm{it}}(\Sigma; B))}{\Delta; \Gamma \vdash \mathrm{it}\ \langle\omega\rangle\ M\ \langle\Omega\rangle : \langle\omega\rangle(B)}\,\mathsf{TpIt}, \qquad \mathrm{dom}(\omega) = \mathcal{I}(\Sigma; B)$$

$$\frac{}{\Delta; \Gamma \vdash \cdot : \langle\omega\rangle(\cdot)}\,\mathsf{TrBase} \qquad \frac{\Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma) \qquad \Delta; \Gamma \vdash M : \langle\omega\rangle(B')}{\Delta; \Gamma \vdash (\Omega \mid c \mapsto M) : \langle\omega\rangle(\Sigma, c : B')}\,\mathsf{TrInd}$$

Example 9. In Example 7 we defined $\mathrm{cntvar} = \lambda x : \Box\mathrm{exp}.\,\mathrm{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle$ where

$$\begin{aligned}
\omega &= \mathrm{exp} \mapsto \Box\mathrm{nat}\\
\Omega &= \mathrm{app} \mapsto \mathrm{plus}, \mathrm{lam} \mapsto \lambda f : \Box\mathrm{nat} \to \Box\mathrm{nat}.\, f\ (\mathrm{box}\ (\mathrm{s}\ \mathrm{z}))\\
\mathcal{S}_{\mathrm{it}}(\Sigma; \mathrm{exp}) &= \mathrm{app} : \mathrm{exp} \to (\mathrm{exp} \to \mathrm{exp}), \mathrm{lam} : (\mathrm{exp} \to \mathrm{exp}) \to \mathrm{exp}
\end{aligned}$$

Under the assumption that $\mathrm{plus} : \Box\mathrm{nat} \to (\Box\mathrm{nat} \to \Box\mathrm{nat})$ it is easy to see that

(1) $\cdot; x : \Box\mathrm{exp} \vdash \lambda f : \Box\mathrm{nat} \to \Box\mathrm{nat}.\, f\ (\mathrm{box}\ (\mathrm{s}\ \mathrm{z})) : (\Box\mathrm{nat} \to \Box\mathrm{nat}) \to \Box\mathrm{nat}$
by TpLam, etc.

(2) $\cdot; x : \Box\mathrm{exp} \vdash \Omega : \langle\omega\rangle(\mathcal{S}_{\mathrm{it}}(\Sigma; \mathrm{exp}))$ by TrBase, Ass., (1)

(3) $\cdot; x : \Box\mathrm{exp} \vdash x : \Box\mathrm{exp}$ by TpVarReg

(4) $\cdot; x : \Box\mathrm{exp} \vdash \mathrm{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle : \Box\mathrm{nat}$ by TpIt from (3) (2)

(5) $\cdot; \cdot \vdash \mathrm{cntvar} : \Box\mathrm{exp} \to \Box\mathrm{nat}$ by TpLam from (4)

Applying a term replacement must be restricted to canonical forms in order to preserve types. Fortunately, our type system guarantees that the subject of an iteration can be converted to canonical form. Applying a replacement then transforms a canonical form V of type B into a well-typed object $\langle\omega; \Omega\rangle(V)$ of type $\langle\omega\rangle(B)$. We call this operation *elimination*. It is defined along the structure of V.

Definition 6 Elimination.

$$\begin{aligned}
\langle\omega; \Omega\rangle(c) &= \begin{cases} M & \text{if } \Omega(c) = M \\ c & \text{otherwise} \end{cases} && (\mathsf{ElConst})\\
\langle\omega; \Omega\rangle(x) &= \Omega(x) && (\mathsf{ElVar})\\
\langle\omega; \Omega\rangle(\lambda x : B.\, V) &= \lambda u : \langle\omega\rangle(B).\, \langle\omega; \Omega \mid x \mapsto u\rangle(V) && (\mathsf{ElLam})\\
\langle\omega; \Omega\rangle(V_1\ V_2) &= \langle\omega; \Omega\rangle(V_1)\ \langle\omega; \Omega\rangle(V_2) && (\mathsf{ElApp})
\end{aligned}$$

The term resulting from elimination might, of course, contain redices and must itself be evaluated to obtain a final value. Thus we obtain the following evaluation rule for iteration.

Definition 7 Evaluation judgment. (extending Definition 4)

$$\frac{\Psi \vdash M \hookrightarrow \mathrm{box}\ M' : \Box B \qquad \cdot \vdash M' \Uparrow V' : B \qquad \Psi \vdash \langle\omega; \Omega\rangle(V') \hookrightarrow V : \langle\omega\rangle(B)}{\Psi \vdash \mathrm{it}\ \langle\omega\rangle\ M\ \langle\Omega\rangle \hookrightarrow V : \langle\omega\rangle(B)}\ \mathsf{EvIt}$$

Example 10. In Example 7, the evaluation of $\mathrm{cntvar}\ (\mathrm{box}\ (\mathrm{lam}\ (\lambda x : \mathrm{exp}.\, x)))$ yields $\mathrm{box}\ (\mathrm{s}\ \mathrm{z})$ because

$(1)\ \cdot \vdash \text{cntvar} \hookrightarrow \text{cntvar} : \Box\text{exp} \to \Box\text{nat}$ by EvLam

$(2)\ \cdot \vdash \text{box}\ (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x)) \hookrightarrow \text{box}\ (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x)) : \Box\text{exp}$ by EvBox

$(3)\ \cdot \vdash \text{lam}\ (\lambda x\!:\!\text{exp}.\, x) \Uparrow \text{lam}\ (\lambda x\!:\!\text{exp}.\, x) : \text{exp}$ by EcAtomic, etc.

$(4)\ \langle \omega; \Omega \rangle (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x))$
$\quad = (\lambda f\!:\!\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s z})))\ (\lambda x'\!:\!\Box\text{nat}.\, x')$ by elimination

$(5)\ \cdot \vdash \langle \omega; \Omega \rangle (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x)) \hookrightarrow \text{box}\ (\text{s z}) : \Box\text{nat}$ by EvApp, etc.

$(6)\ \cdot \vdash \text{it}\ \langle \omega \rangle\ (\text{box}\ (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x)))\ \langle \Omega \rangle \hookrightarrow \text{box}\ (\text{s z}) : \Box\text{nat}$
by EvIt from (2) (3) (5)

$(7)\ \cdot \vdash \text{cntvar}\ (\text{box}\ (\text{lam}\ (\lambda x\!:\!\text{exp}.\, x))) \hookrightarrow \text{box}\ (\text{s z}) : \Box\text{nat}$
by EvApp from (1) (2) (6)

The reader is invited to convince himself that this operational semantics yields the expected results also on the other examples of this section.

Our calculus also contains a **case** construct whose subject may be of type $\Box B$ for arbitrary pure B. It allows us to distinguish cases based on the intensional structure of the subject. For example, we can test if a given (parametric!) function is the identity or not. The typing rules and operational semantics for **case** are similar, but simpler than those for iteration. We therefore elide it here and refer the interested reader to [6].

5 Meta-Theory

The goal of this subsection is to show that the modal λ-calculus obeys the type preservation property and that it is a conservative extension of the simply typed λ-calculus defined in Section 2. We prove this by Tait's method, often called an argument by *logical relations*. After defining the logical relations we then prove the canonical form theorem for the modal λ-calculus which guarantees that every well-typed object eventually evaluates to a canonical form. The type preservation and the conservative extension property follow directly from this theorem.

We now begin the meta-theoretical discussion with the definition of the logical relation. Due to the lazy character of the modal λ-calculus, the interpretation of a type A is twofold: On the one side we would like it to contain all canonical forms of type A, on the other all objects *evaluating* to a canonical form. This is why we introduce two mutual dependent logical relations: In a context Ψ, $[\![A]\!]$ represents the set of objects evaluating to a value being itself an element of $|A|$. For the definition of the logical relation we require the notion of context extension: $\Psi' \geq \Psi$ holds if every declaration in Ψ also occurs in Ψ'.

Definition 8 Logical relation.

$\Psi \vdash M \in [\![A]\!]$ iff $\cdot; \Psi \vdash M : A$ and $\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash V \in |A|$

$\Psi \vdash V \in |A|$ iff

Case: $A = a$ and $\Psi \vdash V \Uparrow a$

Case: $A = A_1 \to A_2$ and either

Case: $V = \lambda x : A_1 . M$ and for all $\Psi' \geq \Psi$: $\Psi' \vdash V' \in |A_1|$ implies $\Psi' \vdash [V'/x](M) \in [\![A_2]\!]$

or

Case: $\Psi \vdash V \downarrow A_1 \to A_2$ and for all $\Psi' \geq \Psi$: $\Psi' \vdash V' \Uparrow A_1$ implies $\Psi' \vdash V\ V' \in |A_2|$

Case: $A = A_1 \times A_2$ and $V = \langle M_1, M_2 \rangle$ and $\Psi \vdash M_1 \in [\![A_1]\!]$ and $\Psi \vdash M_2 \in [\![A_2]\!]$

Case: $A = \Box A'$ and $V = \text{box } M$ and $\cdot \vdash M \in [\![A']\!]$

Since the operational semantics introduced in Section 4 depends on typing information, we must make sure that only well-typed objects are contained in the logical relation. To do so we require that every object $M \in [\![A]\!]$ has type A. As a side effect of this definition the type preservation property is a direct consequence of the canonical form theorem. The proof of the canonical form theorem is split into two parts. In the first part we prove that every element in $[\![A]\!]$ evaluates to a canonical form, in the second part we show that every well-typed object of type A is contained in the logical relation $[\![A]\!]$.

A direct proof of the first property will fail, hence we must generalize its formulation which we can now prove by mutual induction.

Lemma 9 Logical relations and canonical forms.

1. *If* $\Psi \vdash M \in [\![B]\!]$ *then* $\Psi \vdash M \Uparrow V : B$
2. *If* $\Psi \vdash V \downarrow B$ *then* $\Psi \vdash V \in |B|$

The goal of the second part is to show that every well-typed object is in the logical relation, that is, we want to prove that if $\cdot ; \Psi \vdash M : A$ then $\Psi \vdash M \in [\![A]\!]$.

It turns out that we cannot prove this property directly by induction over the structure of the typing derivation, either. The reason is that the context Ψ might grow during the derivation and it may also not remain pure. From the definition of the typing relation it also follows quite directly that Δ need not remain empty. Hence we generalize this property by considering a typing derivation $\Delta ; \Gamma \vdash M : A$ and a substitution $(\theta ; \varrho)$ which maps the variables defined in $\Delta ; \Gamma$ into objects satisfying the logical relation to show that $\Psi \vdash [\theta ; \varrho](M) \in [\![A]\!]$. The objects in θ — which are only substituting modal variables — might not yet be evaluated due to the lazy character of unobservable objects. On the other hand it is safe to assume that objects in ϱ — which substitute for variables in Γ — are already evaluated since function application follows a call-by-value discipline. To make this more precise we define a logical relation for contexts $\Psi \vdash \theta ; \varrho \in [\Delta ; \Gamma]$ iff $\vdash \theta \in [\![\Delta]\!]$ and $\Psi \vdash \varrho \in |\Gamma|$ which are defined as follows:

Definition 10 Logical relation for contexts.

$\vdash \theta \in [\![\Delta]\!]$ iff $\Delta = \cdot$ implies $\theta = \cdot$

and $\Delta = \Delta', x : A$ implies $\theta = \theta', M/x$ and $\cdot \vdash M \in [\![A]\!]$ and $\vdash \theta' \in [\![\Delta']\!]$

$\Psi \vdash \varrho \in |\Gamma|$ iff $\Gamma = \cdot$ implies $\varrho = \cdot$

and $\Gamma = \Gamma', x : A$ implies $\varrho = \varrho', V/x$ and $\Psi \vdash V \in |A|$ and $\Psi \vdash \varrho' \in |\Gamma'|$

We can now formulate and prove

Lemma 11 Typing and logical relations.

If $\Delta; \Gamma \vdash M : A$ *and* $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ *then* $\Psi \vdash [\theta; \varrho](M) \in [\![A]\!]$

The proof of this lemma is rather difficult. Due to the restrictions in length we are not presenting any details here, but refer the interested reader to [6]. Now, an easy inductive argument using Lemma 9 shows that the identity substitution $(\cdot, \mathrm{id}_\Psi)$, which maps all variables defined in Ψ to themselves, indeed lies within the logical relation $[\cdot; \Psi]$. The soundness of typing is hence an immediate corollary of Lemma 11.

Theorem 12 Soundness of typing.

If $\cdot; \Psi \vdash M : A$ *then* $\Psi \vdash M \in [\![A]\!]$.

This theorem together with Lemma 9 guarantees that terms of pure type evaluate to a canonical form V.

Theorem 13 Canonical form theorem.

If $\cdot; \Psi \vdash M : B$ *then* $\Psi \vdash M \Uparrow V : B$ *for some* V.

Type preservation now follows easily: We need to show that the evaluation result of a well-typed object possesses the same type A. From Lemma 11 we obtain that M lies within the logical relation $[\![A]\!]$, which guarantees that M evaluates to a term V, also in the logical relation. Further, a simple induction over the structure of an evaluation shows the values are unique and have the right type.

Theorem 14 Type preservation.

If $\cdot; \Psi \vdash M : A$ *and* $\Psi \vdash M \hookrightarrow V : A$ *then* $\cdot; \Psi \vdash V : A$.

On the basis of the canonical form theorem, we can now prove the main result of the paper: Our calculus is a conservative extension of the simply typed λ-calculus $\lambda^{\rightarrow}$ from Section 2. Let M be an object of pure type B, with free variables from a pure context Ψ. M itself need not be pure but rather some term in the modal λ-calculus including iteration and case. We have seen that M has a canonical form V, and an immediate inductive argument shows that V must be a term in the simply typed λ-calculus.

Theorem 15 Conservative extension.

If $\cdot; \Psi \vdash M : B$ *then* $\Psi \vdash M \Uparrow V : B$ *and* $\Psi \vdash V \Uparrow B$.

6 Conclusion and Future Work

We have presented a calculus for primitive recursive functionals over higher-order abstract syntax which guarantees that the adequacy of encodings remains intact. The requisite conservative extension theorem is technically deep and requires a careful system design and analysis of the properties of a modal operator $\Box$ and its interaction with function definition by iteration and cases. To our knowledge, this is the first system in which it is possible to safely program functionally with higher-order abstract syntax representations. It thus complements and refines the logic programming approach to programming with such representations [17, 21].

Our work was inspired by Miller's system [15], which was presented in the context of ML. Due to the presence of unrestricted recursion and the absence of a modal operator, Miller's system is computationally adequate, but has a much weaker meta-theory which would not be sufficient for direct use in a logical framework. The system of Meijer and Hutton [14] and its refinement by Fegaras and Sheard [8] are also related in that they extend primitive recursion to encompass functional objects. However, they treat functional objects extensionally, while our primitives are designed so we can analyze the internal structure of λ-abstractions directly. Fegaras and Sheard also note the problem with adequacy and design more stringent type-checking rules in Section 3.4 of [8] to circumvent this problem. In contrast to our system, their proposal does not appear to have a logical interpretation. Furthermore, they neither claim nor prove type preservation or an appropriate analogue of conservative extension—critical properties which are not obvious in the presence of their internal type tags and **Place** constructor.

Our system is satisfactory from the theoretical point of view and could be the basis for a practical implementation. Such an implementation would allow the definition of functions of arbitrary types, while data constructors are constrained to have pure type. Many natural functions over higher-order representations turn out to be directly definable (e.g., one-step parallel reduction or conversion to de Bruijn indices), others require explicit counters to guarantee termination (e.g., multi-step reduction or full evaluation). On the other hand, it appears that some natural *algorithms* (e.g., a structural equality check which traverses two expressions simultaneously) are not implementable, even though the underlying function is certainly definable (e.g., via a translation to de Bruijn indices). For larger applications, writing programs by iteration becomes tedious and error-prone and a pattern-matching calculus such as employed in ALF [2] or proposed by Jouannaud and Okada [11] seems more practical. Our informal notation in the examples provides some hints what concrete syntax one might envision for an implementation along these lines.

The present paper is a first step towards a system with dependent types in which proofs of meta-logical properties of higher-order encodings can be expressed directly by dependently typed, total functions. The meta-theory of such a system appears to be highly complex, since the modal operators necessitate a *let box* construct which, *prima facie*, requires commutative conversions. Martin Hofmann[3] has proposed a semantical explanation for our iteration operator which has led him to discover an equational formulation of the laws for iteration. This may be the critical insight required for a dependently typed version of our calculus. A similar formulation of these laws is used in [7] for the treatment of recursion. We also plan to reexamine applications in the realm of functional programming [15, 8] and related work on reasoning about higher-order abstract syntax with explicit induction [5, 4] or definitional reflection [13].

Acknowledgments. The work reported here took a long time to come to fruition, largely due to the complex nature of the technical development. During this time we have discussed various aspects of higher-order abstract syntax, iteration, and induction with too many people to acknowledge them individually. Special thanks go to Gérard Huet and Chet Murthy, who provided the original inspiration, and Hao-Chi Wong who helped us understand the nature of modality in this context.

References

1. Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.
2. Thierry Coquand, Bengt Nordström, Jan M. Smith, and Björn von Sydow. Type theory and programming. *Bulletin of the European Association for Theoretical Computer Science*, 52:203–228, February 1994.
3. Rowan Davies and Frank Pfenning. A modal analysis of staged computation. In Jr. Guy Steele, editor, *Proceedings of the 23rd Annual Symposium on Principles of Programming Languages*, pages 258–270, St. Petersburg Beach, Florida, January 1996. ACM Press.
4. Joëlle Despeyroux, Amy Felty, and André Hirschowitz. Higher-order abstract syntax in Coq. In M. Dezani-Ciancaglini and G. Plotkin, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 124–138, Edinburgh, Scotland, April 1995. Springer-Verlag LNCS 902.
5. Joëlle Despeyroux and André Hirschowitz. Higher-order abstract syntax with induction in Coq. In Frank Pfenning, editor, *Proceedings of the 5th International Conference on Logic Programming and Automated Reasoning*, pages 159–173, Kiev, Ukraine, July 1994. Springer-Verlag LNAI 822.
6. Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. Technical Report CMU-CS-96-172, Carnegie Mellon University, September 1996.
7. Thierry Despeyroux and André Hirschowitz. Some theory for abstract syntax and induction. Draft manuscript.
8. Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Proceedings of 23rd Annual Symposium on Principles of Programming Languages*, pages 284–294, St. Petersburg Beach, Florida, January 1996. ACM Press.
9. Kurt Gödel. On an extension of finitary mathematics which has not yet been used. In Solomon Feferman et al., editors, *Kurt Gödel, Collected Works, Volume II*, pages 271–280. Oxford University Press, 1990.

[3] personal communication

10. Robert Harper, Furio Honsell, and Gordon Plotkin. A framework for defining logics. *Journal of the Association for Computing Machinery*, 40(1):143–184, January 1993.
11. Jean-Pierre Jouannaud and Mitsuhiro Okada. A computation model for executable higher-order algebraic specification languages. In Gilles Kahn, editor, *Proceedings of the 6th Annual Symposium on Logic in Computer Science*, pages 350–361, Amsterdam, The Netherlands, July 1991. IEEE Computer Society Press.
12. Lena Magnusson. *The Implementation of ALF—A Proof Editor Based on Martin-Löf's Monomorphic Type Theory with Explicit Substitution.* PhD thesis, Chalmers University of Technology and Göteborg University, January 1995.
13. Raymond McDowell and Dale Miller. A logic for reasoning about logic specifications. Draft manuscript, July 1996.
14. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Proceedings of the 7th Conference on Functional Programming Languages and Computer Architecture*, La Jolla, California, June 1995.
15. Dale Miller. An extension to ML to handle bound variables in data structures: Preliminary report. In *Proceedings of the Logical Frameworks BRA Workshop*, Nice, France, May 1990.
16. Dale Miller. Unification of simply typed lambda-terms as logic programming. In Koichi Furukawa, editor, *Eighth International Logic Programming Conference*, pages 255–269, Paris, France, June 1991. MIT Press.
17. Dale Miller. Abstract syntax and logic programming. In *Proceedings of the First and Second Russian Conferences on Logic Programming*, pages 322–337, Irkutsk and St. Petersburg, Russia, 1992. Springer-Verlag LNAI 592.
18. Bengt Nordström, Kent Petersson, and Jan M. Smith. *Programming in Martin-Löf's Type Theory: An Introduction*, volume 7 of *International Series of Monographs on Computer Science*. Oxford University Press, 1990.
19. Martin Odersky. A functional theory of local names. In *Proceedings of the 21st Annual Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994. ACM Press.
20. Christine Paulin-Mohring. Inductive definitions in the system Coq: Rules and properties. In M. Bezem and J.F. Groote, editors, *Proceedings of the International Conference on Typed Lambda Calculi and Applications*, pages 328–345, Utrecht, The Netherlands, March 1993. Springer-Verlag LNCS 664.
21. Frank Pfenning. Logic programming in the LF logical framework. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.
22. Frank Pfenning and Hao-Chi Wong. On a modal λ-calculus for S4. In S. Brookes and M. Main, editors, *Proceedings of the Eleventh Conference on Mathematical Foundations of Programming Sematics*, New Orleans, Louisiana, March 1995. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.

This article was processed using the LaTeX macro package with LLNCS style