Primitive Recursion for Higher-Order Abstract Syntax

Carsten Schürmann [a,1] Joëlle Despeyroux [b] Frank Pfenning [c,1]

[a] *Yale University, New Haven, CT 06520-8285, USA, carsten@cs.yale.edu*

[b] *INRIA, 06902 Sophia-Antipolis Cedex, France, joelle.despeyroux@sophia.inria.fr*

[c] *Carnegie Mellon University, Pittsburgh, PA 15213, USA, fp@cs.cmu.edu*

Abstract

Higher-order abstract syntax is a central representation technique in logical frameworks which maps variables of the object language into variables of the meta-language. It leads to concise encodings, but is incompatible with functions defined by primitive recursion or proofs by induction.

In this paper we propose an extension of the simply-typed lambda-calculus with iteration and case constructs which preserves the adequacy of higher-order abstract syntax encodings. The well-known paradoxes are avoided through the use of a modal operator which obeys the laws of S_4. In the resulting calculus many functions over higher-order representations can be expressed elegantly. Our central technical result, namely that our calculus is conservative over the simply-typed lambda-calculus, is proved by a rather complex argument using logical relations.

We view our system as an important first step towards allowing the methodology of LF to be employed effectively in systems based on induction principles such as ALF, Coq, or Nuprl, leading to a synthesis of currently incompatible paradigms.

Key words: typed lambda calculus, higher order abstract syntax, primitive recursion, modal logic

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

[1] This work was sponsored NSF Grant CCR-9303383.

by variables of 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 [Chu40] and Martin-Löf's system of arities [NPS90], has far-reaching consequences for the methodology of logical frameworks. On the 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 [DH94,DFH95], 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 [Göd90]. 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 [HHP93] to be incorporated into systems such as Coq [PM93] or ALF [Mag95].

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 ($\lambda^{\to}$) which we extend to a modal λ-calculus in Section 3. Section 4 then presents the iteration and Section 5 definition by cases. In Section 6 we start with the technical discussion and introduce some auxiliary concepts and derive some basic results. Section 7 shows the proof of the canonical form theorem which is essential for the proof of type preservation (Section 8) and our central result, namely that our system is conservative over $\lambda^{\rightarrow}$ (Section 9). Finally, Section 10 assesses the results, compares some related work, and outlines future work.

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 10). Our formulation of the simply-typed meta-language is standard.

$$\begin{array}{lrcl}
\text{Pure types:} & B & ::= & a \mid B_1 \to B_2 \\
\text{Objects:} & M & ::= & x \mid c \mid \lambda x{:}B.\, M \mid M_1\ M_2 \\
\text{Contexts:} & \Psi & ::= & \cdot \mid \Psi, x : B \\
\text{Signatures:} & \Sigma & ::= & \cdot \mid \Sigma, a : \text{type} \mid \Sigma, c : B
\end{array}$$

We use a for type constants, c for object constants and x for variables. We assume that constants and variables are declared at most once in a signature and context, respectively. As usual, we apply tacit renaming of bound variables to maintain this assumption, and to guarantee capture-avoiding substitution. The typing judgments for objects and signatures are standard. Constants must be declared before they are used. In this paper we can assume the signature Σ to be always well-formed and fixed, and hence omit it from the various judgments.

Definition 2.1 (Typing judgment) *$\Psi \vdash M : B$ is defined by the following rules.*

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

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

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

Example 2.2 (Natural numbers)

$$\begin{array}{rll} & \mathrm{nat} & : \mathrm{type} \\ \ulcorner 0 \urcorner = \mathrm{z} & \mathrm{z} & : \mathrm{nat} \\ \ulcorner n+1 \urcorner = \mathrm{s}\, \ulcorner n \urcorner & \mathrm{s} & : \mathrm{nat} \to \mathrm{nat} \end{array}$$

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

Example 2.3 (Untyped λ-expressions) $e ::= x \mid \mathbf{lam}\ x.e \mid e_1 @ e_2$

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

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 [HHP93], we call these objects *canonical* as defined by the following two judgments, and denote them with V.

Definition 2.4 (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)*

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

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

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 2.3 (untyped λ-expressions) this is expressed by the following property, which is proved by simple inductions.

Example 2.5 (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 $\lambda^{\rightarrow}$ 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 3.3, 4.30, and 5.15) 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 2.3 are "lam" of type $(\mathrm{exp} \rightarrow \mathrm{exp}) \rightarrow \mathrm{exp}$ and "app" of type $\mathrm{exp} \rightarrow (\mathrm{exp} \rightarrow \mathrm{exp})$. 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 definition by cases or an iterator, then not every well-typed object of type exp has a canonical form. For example,

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

has type 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 $\mathrm{exp} \rightarrow \mathrm{exp}$ which cannot be converted to a function in $\lambda^{\rightarrow}$. 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* $\mathrm{exp} \rightarrow \mathrm{exp}$ whose elements are convertible to the form $\lambda x : \mathrm{exp}.\, E$ where E is built only from the constructors app, lam, and the variable x, from the *primitive recursive function space* exp $\Rightarrow$ exp which is intended to encompass functions defined through case distinction and iteration. This separation can be achieved by using a modal operator: exp $\to$ exp will continue to contain only the parametric functions, while exp $\Rightarrow$ exp $= (\Box\text{exp}) \to$ exp contains the primitive recursive functions.

The representation technique of higher-order abstract syntax works exactly as before. That is, the representation types for objects only contain parametric function spaces, and the notion of definitional equality on such objects continues to be just $\beta\eta$-conversion. Therefore, we treat functions of pure type *intensionally*, while functions with types containing $\Box$ are treated *extensionally*. We are not considering their form to be observable, only their input/output behavior. Similarly for products which are not used for object representation. As a result, there is no need to define a notion of definitional equality on impure types. In fact, the natural notion of equality on impure types is extensional and defined by induction over the structure of the types. For example, we would consider two functions f and g of type exp $\Rightarrow$ exp to be (extensionally) equal if $f\,M$ and $g\,M$ have the same canonical form for all canonical forms M of type exp. We have not studied this notion of equality, since it does not play a role in our applications. At this point in our development it is not obvious that this view of our λ-calculus is tenable; eventually it is justified by the conservative extension theorem (Theorem 9.2).

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 [DP96].

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 easily 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 convenient 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 [DP96] and goes back to [PW95]. The language of types includes the pure types from the

$$\frac{\Gamma(x) = A}{\Delta;\Gamma \vdash x : A}\,\mathsf{TpVarR} \qquad \frac{\Delta(x) = A}{\Delta;\Gamma \vdash x : A}\,\mathsf{TpVarM} \qquad \frac{\Sigma(c) = B}{\Delta;\Gamma \vdash c : B}\,\mathsf{TpCon}$$

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

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

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

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

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

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

simply-typed λ-calculus in Section 2.

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

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. Δ and Γ should be viewed as lists, the variable names they declare must be disjoint.

Definition 3.1 (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 3.2 (Laws of S_4)

$$\begin{aligned}
\text{subst} &: \Box(A_1 \to A_2) \to \Box A_1 \to \Box A_2 \\
&= \lambda f{:}\Box(A_1 \to A_2).\, \lambda x{:}\Box A_1. \\
&\qquad \text{let box } f' = f \text{ in let box } x' = x \text{ in box } (f'\ x') \\
\text{unbox} &: \Box A \to A \\
&= \lambda x{:}\Box A.\, \text{let box } x' = x \text{ in } x' \\
\text{boxbox} &: \Box A \to \Box\Box A \\
&= \lambda x{:}\Box A.\, \text{let box } x' = x \text{ in box } (\text{box } x')
\end{aligned}$$

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 former carries the type of the evaluated object which does not need to be pure, whereas the latter is restricted to pure types, since only objects of pure type possess canonical forms. That these type annotations are well-defined follows from the type preservation property we derive later in this paper. Since we evaluate under some λ-abstractions, free variables of pure type declared in Ψ may occur in M and V during evaluation.

Definition 3.3 (Evaluation judgments) *$\Psi \vdash M \hookrightarrow V : A$ and $\Psi \vdash M \Uparrow V : B$ are defined in Figure 2.*

Note that the rules EvApp and EvAt are mutually exclusive, since the evaluation of M_1 in an application $M_1\, M_2$ either yields an atomic term (with a constant or parameter at the head) or a λ-abstraction.

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.

$$\frac{\Psi \vdash M \hookrightarrow V : a}{\Psi \vdash M \Uparrow V : a}\,\mathsf{EcAt} \qquad \frac{\Psi, x : B_1 \vdash M\ x \Uparrow V : B_2}{\Psi \vdash M \Uparrow \lambda x{:}B_1.\,V : B_1 \to B_2}\,\mathsf{EcArr}$$

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

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

$$\frac{\begin{array}{c}\Psi \vdash M_1 \hookrightarrow \lambda x{:}A_2.\,M_1' : A_2 \to A_1 \\ \Psi \vdash M_2 \hookrightarrow V_2 : A_2\end{array} \qquad \Psi \vdash [V_2/x](M_1') \hookrightarrow V : A_1}{\Psi \vdash M_1\ M_2 \hookrightarrow V : A_1}\,\mathsf{EvApp}$$

$$\frac{\Psi \vdash M_1 \hookrightarrow V_1 : B_2 \to B_1 \qquad \Psi \vdash V_1 \downarrow B_2 \to B_1 \qquad \Psi \vdash M_2 \Uparrow V_2 : B_2}{\Psi \vdash M_1\ M_2 \hookrightarrow V_1\ V_2 : B_1}\,\mathsf{EvAt}$$

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

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

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

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

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

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

4.1 Examples

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 4.1 (Addition) The usual type of addition is nat $\to$ nat $\to$ 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 $\Box$nat $\to$ nat $\to$ nat and $\Box$nat $\to$ $\Box$nat $\to$ $\Box$nat.

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

Note that this definition cannot be assigned type nat $\to$ nat $\to$ nat or $\Box$nat $\to$ nat $\to$ $\Box$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* [FS96]. In the case of natural numbers, we replace z : nat by a term $M_z : A$ and s : nat $\to$ nat by a function $M_s : A \to A$. 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 4.2 (Addition via iteration) Addition from Example 4.1 can be formulated in a number of ways with an explicit iteration operator. The simplest one:

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

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

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

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 $\nu x.M$) and extending the function definition dynamically to encompass the new parameters (written as "where $f(x) = M$"). A similar idea has led Odersky to define $\lambda\nu$, an extension of the λ-calculus by local names [Ode94].

Example 4.3 (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 2.3. It can be assigned type $\Box\mathrm{exp} \to \Box\mathrm{nat}$.

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

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

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

For example, after β-reduction and replacement the term

$$\mathrm{cntvar}\ (\mathrm{box}\ (\mathrm{lam}\ (\lambda x{:}\mathrm{exp}.\, \mathrm{app}\ x\ x)))$$

reduces to

$$(\lambda f{:}\Box\mathrm{nat} \to \Box\mathrm{nat}.\, f\ (\mathrm{box}\ (\mathrm{s\ z})))\ (\lambda n{:}\Box\mathrm{nat}.\, \mathrm{plus}\ n\ n)$$

which can in turn be β-reduced to plus (box (s z)) (box (s z)) and finally to the expected answer box (s (s z)).

Note that our operational semantics (see Definition 4.30) goes through different intermediate steps than the sequence above, but leads to the same result. Note also how replacement changes the types and possibly the names of bound variables (from $x : \mathrm{exp}$ to $n : \Box\mathrm{nat}$) in the canonical form to guarantee type preservation.

Example 4.4 (Counting abstractions) The function below counts the number of occurrences of λ-abstractions in an expression. It also has type $\Box\mathrm{exp} \to \Box\mathrm{nat}$.

$$\begin{aligned}
&\mathrm{cntlam}\ (\mathrm{app}\ e_1\ e_2) &&= \mathrm{plus}\ (\mathrm{cntlam}\ e_1)\ (\mathrm{cntlam}\ e_2)\\
&\mathrm{cntlam}\ (\mathrm{lam}\ e) &&= \mathrm{s}\ (\nu x.\, \mathrm{cntlam}\ (e\ x)\ \mathrm{where}\ \mathrm{cntlam}\ x = \mathrm{z})
\end{aligned}$$

Its representation as an iteration follows the same ideas as above.

$$\begin{aligned}
\mathrm{cntlam}\ &:\ \Box\mathrm{exp} \to \Box\mathrm{nat}\\
&= \lambda x : \Box\mathrm{exp}.\, \mathrm{it}\ \langle \mathrm{exp} \mapsto \Box\mathrm{nat}\rangle\ x\\
&\qquad \langle\ \mathrm{app} \mapsto \lambda n_1 : \Box\mathrm{nat}.\, \lambda n_2 : \Box\mathrm{nat}.\, \mathrm{plus}\ n_1\ n_2\\
&\qquad \mid \mathrm{lam} \mapsto \lambda f : \Box\mathrm{nat} \to \Box\mathrm{nat}.\\
&\qquad\qquad\qquad \mathrm{let\ box}\ m = f\ (\mathrm{box\ z})\ \mathrm{in\ box}\ (\mathrm{s}\ m)\rangle
\end{aligned}$$

Example 4.5 (First order logic) First order formulas $F ::= \forall x.\, F \mid F_1 \supset F_2 \mid t_1 = t_2$ and terms t are represented as canonical objects of type o and type i, respectively, over the signature which includes the following declarations.

$$\begin{aligned}
\ulcorner \forall x.\, F \urcorner &= \mathrm{forall}\ (\lambda x : \mathrm{i}.\, \ulcorner F \urcorner) && \mathrm{forall} : (\mathrm{i} \to \mathrm{o}) \to \mathrm{o}\\
\ulcorner F_1 \supset F_2 \urcorner &= \mathrm{impl}\ \ulcorner F_1 \urcorner\ \ulcorner F_2 \urcorner && \mathrm{impl}\ : \mathrm{o} \to \mathrm{o} \to \mathrm{o}\\
\ulcorner t_1 = t_2 \urcorner &= \mathrm{eq}\ \ulcorner t_1 \urcorner\ \ulcorner t_2 \urcorner && \mathrm{eq}\quad : \mathrm{i} \to \mathrm{i} \to \mathrm{o}
\end{aligned}$$

To count the number of equality tests, we can specify cnteq with type $\mathrm{i} \to \Box\mathrm{o} \to \Box\mathrm{nat}$ as follows. We require an argument term t in order to instantiate the universal quantifier (since we did not assume any constants of type i).

$$\begin{aligned}
&\mathrm{cnteq}\ t\ (\mathrm{forall}\ F) &&= \mathrm{cnteq}\ t\ (F\ t)\\
&\mathrm{cnteq}\ t\ (\mathrm{impl}\ F_1\ F_2) &&= \mathrm{plus}\ (\mathrm{cnteq}\ t\ F_1)\ (\mathrm{cnteq}\ t\ F_2)\\
&\mathrm{cnteq}\ t\ (\mathrm{eq}\ t_1\ t_2) &&= \mathrm{box}\ (\mathrm{s\ z})
\end{aligned}$$

A representation of cnteq in the modal λ-calculus has the form:

$$\begin{aligned}
\text{cnteq} \ &: \ \text{i} \to \Box\text{o} \to \Box\text{nat} \\
&= \lambda t{:}\text{i}.\, \lambda F{:}\Box\text{o}.\, \text{it}\ \langle \text{o} \mapsto \Box\text{nat} \rangle\ F \\
&\qquad \langle\, \text{forall} \mapsto \lambda f{:}\text{i} \to \Box\text{nat}.\, (f\ t) \\
&\qquad \mid \text{impl} \mapsto \text{plus} \\
&\qquad \mid \text{eq} \mapsto \lambda t_1{:}\text{i}.\, \lambda t_2{:}\text{i}.\, \text{box}\ (\text{s}\ \text{z}) \rangle
\end{aligned}$$

Example 4.6 (Booleans) Boolean values $b ::= \top \mid \bot$ can be represented as objects of type bool over the signature which includes the following declaration:

$$\begin{array}{ll}
\ulcorner \top \urcorner = \text{true} \qquad & \text{true} : \text{bool} \\
\ulcorner \bot \urcorner = \text{false} \qquad & \text{false} : \text{bool}
\end{array}$$

Informally we can represent the Boolean operation “and” as follows. We must require all argument and all result types are boxed, because the result of and will be used as subject for another case distinction.

$$\begin{aligned}
\text{and true } B_2 \ &= B_2 \\
\text{and false } B_2 &= \text{false}
\end{aligned}$$

A formal representation of “and” is then as follows:

$$\begin{aligned}
\text{and} \ &: \ \Box\text{bool} \to \Box\text{bool} \to \Box\text{bool} \\
&= \lambda B_1{:}\Box\text{bool}.\, \lambda B_2{:}\Box\text{bool}. \\
&\qquad \text{it}\ \langle \text{bool} \mapsto \Box\text{bool} \rangle\ B_1 \\
&\qquad \langle\, \text{true} \mapsto B_2 \mid \text{false} \mapsto \text{box false} \rangle
\end{aligned}$$

Example 4.7 (Constant test) Below we define a function which returns true if a given functional object of type exp $\to$ exp (see Example 2.3) is constant with respect to the first argument.

$$\begin{aligned}
&\text{const } (\lambda x{:}\text{exp. lam } (\lambda y{:}\text{exp. } E\ x\ y)) \\
&\qquad\qquad = \nu y.\, \text{const } (\lambda x{:}\text{exp. } E\ x\ y) \qquad \text{where const } (\lambda x{:}\text{exp. } y) = \text{true} \\
&\text{const } (\lambda x{:}\text{exp. app } (E_1\ x)\ (E_2\ x)) \\
&\qquad\qquad = \text{and } (\text{const } (\lambda x{:}\text{exp. } E_1\ x))\ (\text{const } (\lambda x{:}\text{exp. } E_2\ x)) \\
&\text{const } \lambda x{:}\text{exp. } x = \text{false}
\end{aligned}$$

The representation of const has type $\Box(\text{exp} \rightarrow \text{exp}) \rightarrow \Box\text{bool}$.

$$\begin{aligned}
\text{const} \ &: \ \Box(\text{exp} \rightarrow \text{exp}) \rightarrow \Box\text{bool} \\
&= \lambda F{:}\Box(\text{exp} \rightarrow \text{exp}).\ \text{it } \langle \text{exp} \mapsto \Box\text{bool}\rangle\ F \\
&\qquad\qquad\qquad\qquad \langle\, \text{lam} \mapsto \lambda E{:}\Box\text{bool} \rightarrow \Box\text{bool}.\ (E\ (\text{box true})) \\
&\qquad\qquad\qquad\qquad \mid \text{app} \mapsto \text{and}\rangle\ (\text{box false})
\end{aligned}$$

Note how the last case in the informal definition is represented by applying the result of iteration (which will be of type $\Box\text{bool} \rightarrow \Box\text{bool}$) to box false.

Example 4.8 (Translation to de Bruijn representation) Untyped λ-expressions in de Bruijn form $d ::= n \mid \textbf{lam}\ d \mid d_1@d_2$ are represented as canonical objects of type db over the signature which includes the natural numbers and the following declarations.

$$\begin{aligned}
\ulcorner n \urcorner &= \text{var}\ \ulcorner n \urcorner & \text{var} &: \text{nat} \rightarrow \text{db} \\
\ulcorner \textbf{lam}\ d \urcorner &= \text{lm}\ \ulcorner d \urcorner & \text{lm} &: \text{db} \rightarrow \text{db} \\
\ulcorner d_1@d_2 \urcorner &= \text{ap}\ \ulcorner d_1 \urcorner\ \ulcorner d_2 \urcorner & \text{ap} &: \text{db} \rightarrow \text{db} \rightarrow \text{db}
\end{aligned}$$

A translation from the higher-order representation to de Bruijn form has type $\Box\text{exp} \rightarrow \text{db}$ and is represented formally in terms of an auxiliary function trans of type $\Box\text{exp} \rightarrow \Box\text{nat} \rightarrow \text{db}$:

$$\begin{aligned}
\text{trans (lam } e)\ n &= \text{lm } (\nu x.\, \text{trans } (e\ x)\ (\text{s } n) \\
&\qquad\qquad \text{where (trans } x\ m) = \text{var (minus } m\ n)) \\
\text{trans (app } e_1\ e_2)\ n &= \text{ap (trans } e_1\ n)\ (\text{trans } e_2\ n) \\
\text{dbtrans } e &= \text{trans } e \text{ z}
\end{aligned}$$

At the top level (when translating a closed λ-expression) we can instantiate trans's second argument with (box z) to obtain a function of type $\Box\mathrm{exp} \to \mathrm{db}$. Assuming functions minus (whose definition we discuss in the next section) and unbox (see Example 3.2), this is implemented by the following iteration.

$$
\begin{aligned}
\mathrm{trans} \;:\;& \Box\mathrm{exp} \to \Box\mathrm{nat} \to \mathrm{db} \\
=\;& \lambda x{:}\Box\mathrm{exp}.\, \mathrm{it}\ \langle \mathrm{exp} \mapsto \Box\mathrm{nat} \to \mathrm{db}\rangle\ x \\
&\quad \langle\, \mathrm{lam} \mapsto \lambda f{:}(\Box\mathrm{nat} \to \mathrm{db}) \to (\Box\mathrm{nat} \to \mathrm{db}). \\
&\qquad\qquad \lambda n{:}\Box\mathrm{nat}.\, \mathrm{lm}\ (f\ \ (\lambda m{:}\Box\mathrm{nat}.\, \mathrm{var}\ (\mathrm{unbox}\ (\mathrm{minus}\ m\ n))) \\
&\qquad\qquad\qquad\qquad (\mathrm{let\ box}\ n' = n\ \mathrm{in\ box}\ (\mathrm{s}\ n'))) \\
&\quad \mid \mathrm{app} \mapsto \lambda f_1{:}\Box\mathrm{nat} \to \mathrm{db}.\, \lambda f_2{:}\Box\mathrm{nat} \to \mathrm{db}. \\
&\qquad\qquad \lambda n{:}\Box\mathrm{nat}.\, \mathrm{ap}\ (f_1\ n)\ (f_2\ n)\rangle \\
\mathrm{dbtrans} \;:\;& \Box\mathrm{exp} \to \mathrm{db} \\
=\;& \lambda x{:}\Box\mathrm{exp}.\, \mathrm{trans}\ x\ (\mathrm{box\ z})
\end{aligned}
$$

We omit here similar definitions of functions for bracket abstraction and translation from higher-order terms to SK combinators. We believe pairs are necessary for defining parallel β-reduction (which is convenient in the proof of the Church-Rosser theorem).

Example 4.9 (Parallel reduction) Parallel reduction is here defined over expressions (from Example 2.3). We state the function first informally:

$$
\begin{aligned}
\mathrm{par}\ (\mathrm{app}\ e_1\ e_2) &= \mathrm{par}'\ e_1\ (\mathrm{par}\ e_2) \\
\mathrm{par}\ (\mathrm{lam}\ e_1) &= \mathrm{lam}\ (\lambda x{:}\mathrm{exp}.\, \nu x'.\, \mathrm{par}\ (e_1\ x') \\
&\qquad\qquad \text{where } \mathrm{par}\ x' = x \text{ and } \mathrm{par}'\ x'\ e_3 = \mathrm{app}\ x\ e_3) \\
\mathrm{par}'\ (\mathrm{app}\ e_1\ e_2)\ e_2' &= \mathrm{app}\ (\mathrm{par}'\ e_1\ (\mathrm{par}\ e_2))\ e_2' \\
\mathrm{par}'\ (\mathrm{lam}\ e_1)\ e_2' &= \nu x.\, \mathrm{par}\ (e_1\ x) \\
&\qquad\qquad \text{where } \mathrm{par}\ x\ =\ e_2' \text{ and } \mathrm{par}'\ x\ e_3 = \mathrm{app}\ e_2'\ e_3
\end{aligned}
$$

The type of par is $\Box\mathrm{exp} \to \mathrm{exp}$; the auxiliary function par$'$ has type $\Box\mathrm{exp} \to \mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp})$.

$$
\begin{array}{rl}
\mathrm{par} & : \Box\mathrm{exp} \to \mathrm{exp} \\
& = \lambda e{:}\Box\mathrm{exp}. \\
& \quad \mathrm{fst}(\,\mathrm{it}\ \langle \mathrm{exp} \mapsto \mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp})\rangle\ e \\
& \qquad \langle\ \mathrm{app} \mapsto\ \lambda e_1{:}\mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp}).\, \lambda e_2{:}\mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp}). \\
& \qquad\qquad\qquad \langle\ (\mathrm{snd}\ e_1)\ (\mathrm{fst}\ e_2), \\
& \qquad\qquad\qquad\quad \lambda e_2'{:}\mathrm{exp}.\,\mathrm{app}\ ((\mathrm{snd}\ e_1)\ (\mathrm{fst}\ e_2))\ e_2'\rangle \\
& \qquad \mid \mathrm{lam} \mapsto\ \lambda e_1{:}(\mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp})) \to (\mathrm{exp} \times (\mathrm{exp} \to \mathrm{exp})). \\
& \qquad\qquad\qquad \langle\ \mathrm{lam}\ (\lambda x{:}\mathrm{exp}.\ \ \mathrm{fst}\ (e_1\ \langle x, \lambda e_3{:}\mathrm{exp}.\,\mathrm{app}\ x\ e_3\rangle)), \\
& \qquad\qquad\qquad\quad \lambda e_2'{:}\mathrm{exp}.\,\mathrm{fst}\ (e_1\ \langle e_2', \lambda e_3{:}\mathrm{exp}.\,\mathrm{app}\ e_2'\ e_3\rangle)\rangle\rangle)
\end{array}
$$

The following example illustrates two concepts: mutually dependent types and iteration over the form of a (parametric!) function (which we already saw in Example 4.7).

Example 4.10 (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 [Mil91]. Normal forms of the untyped λ-calculus $N ::= P \mid \mathbf{lam}\ x.N$ are represented by the type nf with an auxiliary definition for atomic forms $P ::= x \mid P@N$ of type at. In this example the representation 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 subnf : $\Box(\mathrm{at} \to \mathrm{nf}) \to \mathrm{at} \to \mathrm{nf}$ and subat :

$\Box(\mathrm{at} \to \mathrm{at}) \to \mathrm{at} \to \mathrm{at}$.

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

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.

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

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

Example 4.11 (Further mathematical operations) Below we define the multiplication and the exponentiation function which we can informally define as follows:

$$
\begin{aligned}
\mathrm{mult}\ \mathrm{z}\ N &= \mathrm{z} \\
\mathrm{mult}\ (\mathrm{s}\ M)\ N &= \mathrm{plus}\ (\mathrm{mult}\ M\ N)\ N
\end{aligned}
$$

$$
\begin{aligned}
\mathrm{ex}\ M\ \mathrm{z} &= \mathrm{s}\ \mathrm{z} \\
\mathrm{ex}\ M\ (\mathrm{s}\ N) &= \mathrm{mult}\ (\mathrm{ex}\ M\ N)\ M
\end{aligned}
$$

The representation of mult and ex has type $\Box\text{nat} \to \Box\text{nat} \to \Box\text{nat}$.

$$\begin{aligned}
\text{mult} &: \Box\text{nat} \to \Box\text{nat} \to \Box\text{nat} \\
&= \lambda M{:}\Box\text{nat}.\, \lambda N{:}\Box\text{nat}.\ \text{it}\ \langle \text{nat} \mapsto \Box\text{nat}\rangle\ M \\
&\qquad\qquad\qquad\qquad \langle\, \text{z} \mapsto \text{box z} \\
&\qquad\qquad\qquad\qquad \mid \text{s} \mapsto \lambda M'{:}\Box\text{nat}.\, (\text{plus}\ M'\ N)\rangle
\end{aligned}$$

$$\begin{aligned}
\text{ex} &: \Box\text{nat} \to \Box\text{nat} \to \Box\text{nat} \\
&= \lambda M{:}\Box\text{nat}.\, \lambda N{:}\Box\text{nat}.\ \text{it}\ \langle \text{nat} \mapsto \Box\text{nat}\rangle\ N \\
&\qquad\qquad\qquad\qquad \langle\, \text{z} \mapsto \text{box (s z)} \\
&\qquad\qquad\qquad\qquad \mid \text{s} \mapsto \lambda N'{:}\Box\text{nat}.\, (\text{mult}\ M\ N')\rangle
\end{aligned}$$

Example 4.12 (Ackermann's function) Below we define the function which we can informally define as follows:

$$\begin{aligned}
\text{A z} &= \lambda x{:}\text{nat}.\, (\text{s}\ x) \\
\text{A (s}\ n) &= \lambda x{:}\text{nat}.\, (\text{A}\ n)^x\ x
\end{aligned}$$

where $(f^x\ y)$ stands for $\underbrace{(f ... (f}_{x-\text{times}}\ \ y))$. The representation of A has type $\Box\text{nat} \to \Box\text{nat} \to \Box\text{nat}$.

$$\begin{aligned}
\text{A} &: \Box\text{nat} \to \Box\text{nat} \to \Box\text{nat} \\
&= \lambda m{:}\Box\text{nat}.\ \text{it}\ \langle \text{nat} \mapsto \Box\text{nat} \to \Box\text{nat}\rangle\ m \\
&\qquad\qquad \langle\, \text{z} \mapsto \lambda x{:}\Box\text{nat}.\, \text{let box}\ x' = x\ \text{in box (s}\ x') \\
&\qquad\qquad \mid \text{s} \mapsto \lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, \lambda x{:}\Box\text{nat}. \\
&\qquad\qquad\qquad\qquad \text{it}\ \langle \text{nat} \mapsto \Box\text{nat}\rangle\ x\ \langle \text{z} \mapsto x | \text{s} \mapsto f\rangle\rangle
\end{aligned}$$

The following example shows a scheme how to represent primitive recursion over natural numbers using pairs.

Example 4.13 (Primitive recursion over natural numbers) Below we define a general primitive recursive scheme over natural numbers. Let A be the result type of the primitive recursion. For every $N_z : A$ and $N_s : \Box\text{nat} \to$

$A \to A$ we define informally the primitive recursion scheme:

$$\begin{aligned} \mathrm{pr}\ \mathrm{z} &= N_z \\ \mathrm{pr}\ (\mathrm{s}\ m')\ N &= N_s\ m'\ (\mathrm{pr}\ m') \end{aligned}$$

For the representation of pr we use the standard technique of iteration returning a pair. They allow us to recover the structure of the argument in the following way:

$$\begin{aligned} \mathrm{pr}\ :\ & \Box\mathrm{nat} \to A \\ = {} & \lambda m{:}\Box\mathrm{nat}.\,\mathrm{snd} \\ & \quad (\mathrm{it}\ \langle \mathrm{nat} \mapsto \Box\mathrm{nat} \times A\rangle\ m \\ & \quad \langle\, \mathrm{z} \mapsto \langle \mathrm{box}\ \mathrm{z}, N_z\rangle \\ & \quad \mid \mathrm{s} \mapsto \lambda p{:}\Box\mathrm{nat} \times A. \\ & \qquad\quad \langle \mathrm{let}\ \mathrm{box}\ m' = \mathrm{fst}\ p\ \mathrm{in}\ \mathrm{box}\ (\mathrm{s}\ m'), N_s\ (\mathrm{fst}\ p)\ (\mathrm{snd}\ p)\rangle\rangle) \end{aligned}$$

4.2 Formal Discussion

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 4.10).

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

The types being replaced form a type domain, i.e., a set of pairwise different type constants. Since there are no dependencies, the constants do not have to occur in any particular order. As a general convention, we use the order in which they are declared in the signature.

$$\text{Type domains: } \alpha ::= \cdot \mid \alpha, a$$

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, since it can appear in that role in the body of a function. This leads to the introduction of *well-formed* type replacements $\vdash \omega : \alpha$.

Definition 4.14 (Well formed type replacements)

$$\frac{}{\vdash \cdot : \cdot}\ \mathsf{WrBase} \qquad \frac{\vdash \omega : \alpha}{\vdash (\omega \mid a \mapsto A) : (\alpha, a)}\ \mathsf{WrInd}$$

We address now the question of mutual dependency between atomic types by defining 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. We say that type a_1 subordinates type a_2 if objects of the later type can be constructed from objects of the former type.

Objects of pure type B can contain constructors — from the signature — or parameters — introduced locally by λ-abstractions — with the same *target type* as B. The target type refers to the type a fully applied constructor or parameter belongs to. We denote the *target type* of a pure type B by $\tau(B)$.

Definition 4.15 (Target types)

$$\tau(a) := a$$

$$\tau(B_1 \to B_2) := \tau(B_2)$$

Let B be the type of a constructor or parameter and M be an object of type B. The set of other objects from which M can be constructed can be directly extracted from B, namely all objects of the argument types of B — regardless of whether they occur positively or negatively. For a given pure type B we define the type domain $\mathrm{Source}(B)$ as

Definition 4.16 (Source types)

$$\mathit{Source}(a) := \cdot$$

$$\mathit{Source}(B_1 \to B_2) := \mathit{Source}(B_1) \cup \{\tau(B_1)\} \cup \mathit{Source}(B_2)$$

The source of B is the set of all atomic type appearing in B, except its target type. For example 4.10 it is easily verified that the constructor type of ap yields:

$$\mathrm{Source}(\mathrm{at} \to \mathrm{nf} \to \mathrm{at}) = \{\mathrm{at}, \mathrm{nf}\}$$

To view a set as a type domain, we transform it into a list following their order of declaration. To obtain the set of all types on which an atomic type a may depend, we must select a subset of the signature Σ containing all constant declarations with target type a. This set is called a *sub-signature* for a and denoted by $\mathcal{S}(\Sigma; a)$:

Definition 4.17 (Sub-signature)

$$\mathcal{S}(\cdot; a) = \cdot$$

$$\mathcal{S}(\Sigma, c : B; a) = \begin{cases} \mathcal{S}(\Sigma; a), c : B & \textit{if } \tau(B) = a \\ \mathcal{S}(\Sigma; a) & \textit{otherwise} \end{cases}$$

$$\mathcal{S}(\Sigma, a' : \textit{type}; a) = \mathcal{S}(\Sigma; a)$$

In the setting of mutually dependent types, the notion of sub-signature must be extended to capture additional dependencies. Type domains have been introduced to represent the set of all participating atomic types mutually depending on each other. The definition of a *sub-signature over type domains* $\mathcal{S}^*(\Sigma; \alpha)$ follows easily:

Definition 4.18 (Sub-signature over type domains)

$$\mathcal{S}^*(\Sigma; \cdot) = \cdot$$

$$\mathcal{S}^*(\Sigma; \alpha, a) = \mathcal{S}^*(\Sigma; \alpha), \mathcal{S}(\Sigma; a)$$

The subordination relation reflects dependencies between atomic types. The target type and the source types of a declaration contain subordination information: each source type is *subordinate* to the target type.

Definition 4.19 (Immediate subordination relation) *Let B be a pure type.*

$$a <_B a' \textit{ iff } a \in \textit{Source}(B) \textit{ and } a' = \tau(B)$$

The union of all immediate subordination relations induced by a sub-signature Σ yields the static subordination relation. It is called static because it is derived from the fixed signature.

Definition 4.20 (Static subordination relation) *Let Σ be a signature.*

$$a_1 \lhd_\Sigma a_2 \textit{ iff } \Sigma = \Sigma', c : B \textit{ and either } a_1 <_B a_2 \textit{ or } a_1 \lhd_{\Sigma'} a_2$$

The static subordination relation for Example 4.10 is

$$\text{at} \lhd_\Sigma \text{at}, \text{nf} \lhd_\Sigma \text{at}, \text{at} \lhd_\Sigma \text{nf}, \text{nf} \lhd_\Sigma \text{nf}.$$

Sub-signatures are not the only source on which the subordination relation is based. As briefly mentioned above, another source is iteration over functions. Functional subject types can introduce new dependencies into the subordination graph as the following example shows.

Example 4.21 (Higher-order logic) First order logic can be extended to higher order logic by introducing a reification function from formulas to terms. To count the number of equality tests, we extend the subject of iteration defined in Example 4.5 by a new abstraction over the reification function r which has type $\mathrm{o} \to \mathrm{i}$. The introduction of a reification function makes terms and formulas depend mutually on each other. We therefore must distinguish between cnteqi of type $\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{i}) \to \Box\mathrm{nat}$ which counts occurrences of equality tests in terms and cnteqo of type $\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \Box\mathrm{nat}$ which counts them in formulas.

$$\begin{aligned}
&\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, \mathrm{forall}\ (\lambda x{:}\mathrm{i}.\, F\ r\ x)) \\
&\qquad = \nu x.\, (\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, F\ r\ x)) \qquad \text{where } \mathrm{cnteqi}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, x) = \mathrm{z} \\
&\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, \mathrm{impl}\ (F_1\ r)\ (F_2\ r)) \\
&\qquad = \mathrm{plus}\ (\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, F_1\ r))\ (\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, F_2\ r)) \\
&\mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, \mathrm{eq}\ (t_1\ r)\ (t_2\ r)) \\
&\qquad = \mathrm{s}\ (\mathrm{plus}\ (\mathrm{cnteqi}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, t_1\ r))\ (\mathrm{cnteqi}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, t_2\ r))) \\
&\mathrm{cnteqi}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, r\ (F\ r)) = \mathrm{cnteqo}\ (\lambda r{:}\mathrm{o} \to \mathrm{i}.\, F\ r)
\end{aligned}$$

The representation of cnteqo in the modal λ-calculus has the form:

$$\begin{aligned}
\mathrm{cnteqo}\ :\ & \Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \Box\mathrm{nat} \\
= & \ \lambda F{:}\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}).\, \mathrm{it}\ \langle \mathrm{o} \mapsto \Box\mathrm{nat}, \mathrm{i} \mapsto \Box\mathrm{nat}\rangle\ F \\
& \quad \langle\, \mathrm{forall} \mapsto \lambda f{:}\Box\mathrm{nat} \to \Box\mathrm{nat}.\, (f\ (\mathrm{box}\ \mathrm{z})) \\
& \quad \mid \mathrm{impl} \mapsto \mathrm{plus} \\
& \quad \mid \mathrm{eq} \mapsto \lambda m{:}\Box\mathrm{nat}.\, \lambda n{:}\Box\mathrm{nat}. \\
& \qquad\qquad \mathrm{let\ box}\ r = \mathrm{plus}\ m\ n\ \mathrm{in\ box}\ (\mathrm{s}\ r)\rangle
\end{aligned}$$

Clearly the type of the iteration subject must be taken into consideration when defining the general subordination relation. We proceed now by characterizing all those dependencies which arise from the type B of the iteration subject which will lead to the notion of *dynamic* subordination. From the example above we can see that variables occurring in the closed subject of iteration can be interpreted as constructors if we look at the object from a purely syntactical point of view. We call those variables *parameters* and correspondingly their types *parameter types*.

In the next step we define the dynamic subordination relation which can be directly determined from the set of parameter types. We follow the same idea as in the static case: every parameter type in $\mathcal{P}(B)$ induces a new set of dependencies. Closing all these relations we finally arrive at the dynamic subordination relation:

Definition 4.22 (Dynamic subordination relation) *Let B be a pure type.*

$$a_1 \lhd_B a_2 :\Leftrightarrow B = B_1 \to B_2 \textit{ and either } a_1 <_{B_1} a_2 \textit{ or } a_1 \lhd_{B_2} a_2$$

Consider the type $B = (\mathrm{o} \to \mathrm{i}) \to \mathrm{o}$ from the previous example. The dynamic subordination relation is then characterized by $\mathrm{o} \lhd_B \mathrm{i}$. $\mathrm{o} \lhd_B \mathrm{i}$ expresses that objects of type o can be coerced into objects of type i. The mere presence of such an coercion function turns the first order logic from example 4.5 into a higher order logic. Static and dynamic subordination represent local dependencies between atomic types. To obtain the global subordination relation, the union of both must be closed under transitivity.

Definition 4.23 (Global subordination relation) *Let B be a pure type.*

$$\blacktriangleleft_{\Sigma;B} \;:\Leftrightarrow (\lhd_\Sigma \cup \lhd_B)^+$$

Note, that the global subordination relation is not necessarily reflexive. The simplest example for a non-reflexive subordination relation is type bool from Example 4.6. bool is not recursive, hence it doesn't hold that bool $\blacktriangleleft_{\Sigma;B}$ bool. A closer look reveals, that the subordination relation for bool is empty. But it is definitely not the case that bool is. To account for this observation we extend the notion of subordination relation. If $a \trianglelefteq_{\Sigma;B}$ bool holds then objects of type a can occur as objects or subobjects of objects of type bool. We call this the *weak* subordination relation which is obtained by closing the global subordination relation under reflexivity.

Definition 4.24 (Weak subordination relation)

$$\underline{\blacktriangleleft}_{\Sigma;B} \;:\Leftrightarrow (\lhd_\Sigma \cup \lhd_B)^*$$

Mutually dependent types and the notion of subordination are very closely related. In fact, the subordination relation is defined with the purpose to define an equivalence class of mutually dependent types. Static type subordination is built into calculi where inductive types are defined explicitly (such as the Calculus of Inductive Constructions [PM93]); 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 which is enforced in the typing rules for valid signatures (which have omitted). 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.

As we have seen in example 4.21, the dynamic subordination relation implies that terms and formulas depend on each other. Hence, static subordination constitutes only part of the subordination relation. If we follow the paradigm used in Coq we would calculate internally a syntactical definition of the new inductive type, where parameters are defined as real constructors. This has to be done on the fly because as we will see later in the typing rules, the type of the subject of iteration B must be inferred first. It is indeed possible to show the equivalence of both formulations (which we are not going to do here). All type constants which are mutually dependent with $\tau(B)$, written $\mathcal{I}(\Sigma; B)$, form an equivalence class.

Definition 4.25 (Equivalence class of mutually dependent types) *Let B be a type and Σ a signature:*

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

Revisiting Example 4.21 extending first order logic to higher order logic we can calculate the equivalence class $\mathcal{I}(\Sigma; (\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) = \{\mathrm{o}, \mathrm{i}\}$. The sub-signature has then the following form:

$$\mathcal{S}^*(\Sigma; \mathrm{o}, \mathrm{i}) = \mathrm{forall} : (\mathrm{i} \to \mathrm{o}) \to \mathrm{o}, \mathrm{impl} : \mathrm{o} \to \mathrm{o} \to \mathrm{o}, \mathrm{eq} : \mathrm{i} \to \mathrm{i} \to \mathrm{o}$$

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. The replacement application might traverse type constants not defined in ω. This becomes immediately evident when we consider Example 4.8: nat is traversed, but not defined in ω. Also in Example 4.5: i is not defined in ω. But since objects of such strictly subordinated types do not participate in the process of iteration, their types remain unchanged.

Definition 4.26 (Type replacement application) *Let ω be a type replacement:*

$$\langle\omega\rangle(a) := \begin{cases} A \text{ } \textit{if } \omega(a) = A \\ a \text{ } \textit{ otherwise} \end{cases}$$
$$\langle\omega\rangle(B_1 \to B_2) := \langle\omega\rangle(B_1) \to \langle\omega\rangle(B_2)$$

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.

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

The domain of a term replacement is a signature $\mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ containing all constructors whose target type is in $\mathcal{I}(\Sigma; B)$. We extend the notion of objects by

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

and extend the typing rules for iteration. To do so we must introduce a new typing judgment for term replacements Ω: $\Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma)$. Ω is well-typed if it replaces every constant of some signature Σ with some object of the correct type. Note that $\langle\omega\rangle$ is part of the typing judgment and not an operation on signatures.

Definition 4.27 (Typing judgment) *extending Definition 3.1:*

$$\frac{\Delta; \Gamma \vdash M : \Box B \qquad \vdash \omega : \alpha \qquad \Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma')}{\Delta; \Gamma \vdash \textit{it } \langle\omega\rangle \ M \ \langle\Omega\rangle : \langle\omega\rangle(B)} \ \mathsf{TpIt}$$

$$\textit{where } \alpha = \mathcal{I}(\Sigma; B) \textit{ and } \Sigma' = \mathcal{S}^*(\Sigma; \alpha)$$

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

Example 4.28 (Counting variable occurrences) In Example 4.3 we defined $\text{cntvar} = \lambda x{:}\Box\text{exp}.\,\text{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle$ where

$$\begin{aligned}
\omega &= \text{exp} \mapsto \Box\text{nat} \\
\Omega &= \text{app} \mapsto \text{plus} \mid \text{lam} \mapsto \lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s z})) \\
\Sigma' &= \mathcal{S}^*(\Sigma;\ \mathcal{I}(\Sigma;\text{exp})) \\
&= \text{app} : \text{exp} \to (\text{exp} \to \text{exp}), \text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}
\end{aligned}$$

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

$$\begin{array}{lll}
(1) & \cdot; x : \Box\text{exp} \vdash \lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s z})) : (\Box\text{nat} \to \Box\text{nat}) \to \Box\text{nat} & \\
& & \text{by } \mathsf{TpLam}\text{, etc.} \\
(2) & \cdot; x : \Box\text{exp} \vdash \Omega : \langle\omega\rangle(\Sigma') & \text{by } \mathsf{TrBase}\text{, Ass., (1)} \\
(3) & \cdot; x : \Box\text{exp} \vdash x : \Box\text{exp} & \text{by } \mathsf{TpVarR} \\
(4) & \cdot; x : \Box\text{exp} \vdash \text{it}\ \langle\omega\rangle\ x\ \langle\Omega\rangle : \Box\text{nat} & \text{by } \mathsf{TpIt} \text{ from (3) (2)} \\
(5) & \cdot; \cdot \vdash \text{cntvar} : \Box\text{exp} \to \Box\text{nat} & \text{by } \mathsf{TpLam} \text{ from (4)}
\end{array}$$

The formalization of the transformation function of λ-expressions into de Bruijn representation in Example 4.8 is done in the following way. First, the type of the function must be inferred to make explicit which arguments must be boxed and which not. This is mainly determined by the subject of the iteration, here $\Box\text{exp}$. Second, the type replacement ω needs to be specified: $\omega = \text{exp} \mapsto \Box\text{nat} \to \text{db}$. The equivalence class of mutually dependent types $\mathcal{I}(\Sigma;\text{exp}) = \{\text{exp}\}$ already determines the sub-signature:

$$\mathcal{S}^*(\Sigma;\text{exp}) = \text{lam} : (\text{exp} \to \text{exp}) \to \text{exp}, \text{app} : \text{exp} \to (\text{exp} \to \text{exp})$$

and together with the type replacement ω the types occurring in the term replacement. The replacement for lam must be of type

$$((\Box\text{nat} \to \text{db}) \to (\Box\text{nat} \to \text{db})) \to (\Box\text{nat} \to \text{db})$$

and similarly the replacement for app

$$(\Box\text{nat} \to \text{db}) \to (\Box\text{nat} \to \text{db}) \to (\Box\text{nat} \to \text{db}).$$

The iteration itself has hence the type $(\Box\text{nat} \to \text{db})$.

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. 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 rename variables x of type B by new variables x' of type $\langle\omega\rangle(B)$. The definition of a term replacement is extended accordingly.

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

Applying a replacement then transforms a canonical form V of type B into a well-typed object $\langle\omega;\Omega\rangle(V)$ of type $\langle\omega\rangle(B)$ as we will show later in this paper. We call this operation *elimination*. It is defined along the structure of V.

Definition 4.29 (Elimination)

$$\langle\omega;\Omega\rangle(c) = \begin{cases} M & \mathit{if}\ \Omega(c) = M \\ c & \mathit{otherwise} \end{cases} \qquad (\mathsf{ElConst})$$

$$\langle\omega;\Omega\rangle(x) = \Omega(x) \qquad (\mathsf{ElVar})$$

$$\langle\omega;\Omega\rangle(\lambda x\!:\!B.\,V) = \lambda x'\!:\!\langle\omega\rangle(B).\,\langle\omega;\Omega \mid x \mapsto x'\rangle(V) \qquad (\mathsf{ElLam})$$

$$\langle\omega;\Omega\rangle(V_1\ V_2) = \langle\omega;\Omega\rangle(V_1)\ \langle\omega;\Omega\rangle(V_2) \qquad (\mathsf{ElApp})$$

Note that the additional cases in a term replacement do not require additional typing rules since they occur only temporarily during elimination.

Constructors and variables must be mapped to some objects defined in the term replacement Ω. As mentioned above, not all types occurring in the subject type of the iteration object are mutually dependent. This property implies that elimination might encounter constructors which are not defined in the term replacement. In this case we do not replace the constants, as already indicated by the type replacement which leaves those atomic types unchanged. When eliminating a λ-abstraction $\lambda x : B.\,V$, ElLam applies: x, introduced by the λ-abstraction is a parameter which will be renamed to x'. The term replacement must hence be extended by $x \mapsto x'$. The elimination result must then be abstracted over the newly introduced variable x' of type $\langle\omega\rangle(B)$.

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 4.30 (Evaluation judgment) *extending Definition 3.3:*

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

Example 4.31 (Counting variable occurrences) In Example 4.3, the evaluation of $\text{cntvar}\ (\text{box}\ (\text{lam}\ (\lambda x{:}\text{exp}.\, x)))$ yields $\text{box}\ (\text{s z})$ 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 EcAt, etc.

(4) $\langle\omega;\Omega\rangle(\text{lam}\ (\lambda x{:}\text{exp}.\, x))$

$= (\lambda f{:}\Box\text{nat} \to \Box\text{nat}.\, f\ (\text{box}\ (\text{s z})))\ (\lambda x'{:}\Box\text{nat}.\, x')$ 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 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. We discuss the case construct in the next section.

5 Case

Gödel's system T is based on primitive recursion at higher types. For inductive types, primitive recursion can be emulated directly in our system by iteration and pairs. However, it is significantly more difficult to define functions by cases in terms of iterators. Intrinsically, iteration traverses the syntactical structure of a term, whereas case analysis considers only the top-level structure of a term. In simple instances of first-order and higher-order datatypes, we have succeed in emulating case constructs by iteration, which resulted in two basic observations for the general cases. First, it is very expensive to recurse through the structure of a term with the only goal to decide a case expression, and second, the use of iterators requires a significant effort to maintain information about which arguments of a case subject are closed. Therefore, instead of defining it in terms of iteration, we add a new construct for case analysis to the calculus, which avoids unnecessary recursion, and maintains closure information about its arguments. This simplifies the presentation of the examples in this section tremendously.

5.1 Examples

We start with some simple examples, motivating the case operator which is then formally introduced at the end of this section.

Example 5.1 (Comparison) To check if a natural number is greater than 0 we would like to write informally

$$\begin{aligned} \mathrm{gt0}\ m = \mathrm{case}\ m\ \mathrm{of}\ \mathrm{z} &\Rightarrow \mathrm{false} \\ \mid (\mathrm{s}\ m') &\Rightarrow \mathrm{true} \end{aligned}$$

Case distinction is generally triggered by the head constant of the case subject. As for iteration, the subject of case must be always closed. This property derives from the fact that a definition by cases is only complete iff all possible constructors of its subject are covered. The case of $m = \mathrm{z} : \mathrm{nat}$ should trigger $\mathrm{false} : \mathrm{bool}$; otherwise $m = \mathrm{s}\ n$ holds, which should trigger some object M_s. The type of M_s is determined by the type of $\mathrm{s} : \mathrm{nat} \to \mathrm{nat}$: since m is atomic and closed, all parameters to the head constructor in m are also closed. The target type of M_s is bool, justifying $M_s : \Box\mathrm{nat} \to \mathrm{bool}$. Hence $M_s = \lambda n : \Box\mathrm{nat}.\, \mathrm{true}$ formalizes the informal notation.

The case construct can be easily generalized for objects of arbitrary closed atomic types. The challenge in designing a case operator for our system is to extend this generalization to functional types. This is a difficult endeavor because subterms of the case subject are in general not closed since they may contain free parameters, and the system including case should still be conservative over the simply typed λ-calculus.

The case construct in its simplest form is written as "case $\langle A \rangle\ M\ \langle \Xi \rangle$" where M (of type $\Box B$) is the subject of case, and Ξ is a list containing matches for all constructors of type $\tau(B)$. If B is a simple type, A is the result type of the case, and if B is higher-order, then the result type of the case depends on both, A and B.

Example 5.2 (Comparison with case) The greater-than function from example 5.1 can be formulated as follows:

$$\begin{aligned} \mathrm{gt0} \;:\; & \Box\mathrm{nat} \to \mathrm{bool} \\ = \; & \lambda m\!:\!\Box\mathrm{nat}.\, \mathrm{case}\ \langle\mathrm{bool}\rangle\ m\ \langle \mathrm{z} \Rightarrow \mathrm{false} \mid \mathrm{s} \Rightarrow \lambda n\!:\!\Box\mathrm{nat}.\, \mathrm{true}\rangle \end{aligned}$$

Boolean connectives (see Example 4.6) as "not" and "or" can be expressed using iteration (which we have not done) but they can also be expressed using case (as can "and"). We require all argument and result types to be boxed, because the combination of Boolean connectives allows the result of one connective to appear as the argument of another.

Example 5.3 (Boolean operators)

$$\begin{aligned} \mathrm{not}\ B = \;& \mathrm{case}\ B\ \mathrm{of} \\ & \langle\, \mathrm{true} \Rightarrow \mathrm{false} \\ & \mid \mathrm{false} \Rightarrow \mathrm{true}\rangle \end{aligned} \qquad \begin{aligned} \mathrm{or}\ B_1\ B_2 = \;& \mathrm{case}\ B_1\ \mathrm{of} \\ & \langle\, \mathrm{true} \Rightarrow \mathrm{true} \\ & \mid \mathrm{false} \Rightarrow B_2\rangle \end{aligned}$$

The formal representation of the Boolean operations is as follows:

$$\begin{aligned} \mathrm{not} \;:\; & \Box\mathrm{bool} \to \Box\mathrm{bool} \\ = \; & \lambda B\!:\!\Box\mathrm{bool}. \\ & \quad \mathrm{case}\ \langle\Box\mathrm{bool}\rangle\ B \\ & \quad \langle\, \mathrm{true} \Rightarrow \mathrm{box\ false} \\ & \quad \mid \mathrm{false} \Rightarrow \mathrm{box\ true}\rangle \end{aligned} \qquad \begin{aligned} \mathrm{or} \;:\; & \Box\mathrm{bool} \to \Box\mathrm{bool} \to \Box\mathrm{bool} \\ = \; & \lambda B_1\!:\!\Box\mathrm{bool}.\, \lambda B_2\!:\!\Box\mathrm{bool}. \\ & \quad \mathrm{case}\ \langle\Box\mathrm{bool}\rangle\ B_1 \\ & \quad \langle\, \mathrm{true} \Rightarrow \mathrm{box\ true} \\ & \quad \mid \mathrm{false} \Rightarrow B_2\rangle \end{aligned}$$

We continue our presentation with subtraction (which we already assumed to be representable in Example 4.8) where we will use a combination of iteration and case distinction.

Example 5.4 (Subtraction) Among others, the type of subtraction could be $\Box\mathrm{nat} \to \Box\mathrm{nat} \to \Box\mathrm{nat}$. It is informally defined as follows.

$$\begin{aligned} \mathrm{minus}\ m\ \mathrm{z} \quad\; &= m \\ \mathrm{minus}\ m\ (\mathrm{s}\ n') &= \mathrm{case}\ m\ \mathrm{of}\ \mathrm{z} \Rightarrow \mathrm{z} \\ & \qquad \mid (\mathrm{s}\ m') \Rightarrow (\mathrm{minus}\ m'\ n') \end{aligned}$$

Both arguments of minus must be closed, because we use case distinction over the first argument and iteration over the second.

$$
\begin{aligned}
\text{minus} \ &: \ \Box\text{nat} \to \Box\text{nat} \to \Box\text{nat} \\
&= \lambda x\!:\!\Box\text{nat}.\, \lambda y\!:\!\Box\text{nat}.\, \text{it}\ \langle \text{nat} \mapsto (\Box\text{nat} \to \Box\text{nat})\rangle\ y \\
&\qquad \langle\, \text{z} \mapsto \lambda m\!:\!\Box\text{nat}.\, m \\
&\qquad \mid \text{s} \mapsto \lambda n\!:\!(\Box\text{nat} \to \Box\text{nat}). \\
&\qquad\qquad \lambda m\!:\!\Box\text{nat}.\, \text{case}\ \langle \Box\text{nat}\rangle\ m \\
&\qquad\qquad\quad \langle\, \text{z} \Rightarrow \text{box z} \\
&\qquad\qquad\quad \mid \text{s} \Rightarrow \lambda m'\!:\!\Box\text{nat}.\, (n\ m')\rangle\rangle\ x
\end{aligned}
$$

The **case** construct can also be used to distinguish cases over functions. For example, we can test if a given (parametric!) function is the identity or not.

Example 5.5 (Identity test) Below is a function which decides if a parametric function mapping exp to exp is the identity function or not. The function has type $\Box(\text{exp} \to \text{exp}) \to \text{bool}$.

$$
\begin{aligned}
\text{id-test}\ E = \text{case}\ E\ \text{of}\ &\lambda x\!:\!\text{exp}.\, (\text{app}\ (E_1\ x)\ (E_2\ x)) \Rightarrow \text{false} \\
\mid\ &\lambda x\!:\!\text{exp}.\, (\text{lam}\ \lambda y\!:\!\text{exp}.\, E\ x\ y) \Rightarrow \text{false} \\
\mid\ &\lambda x\!:\!\text{exp}.\, x \Rightarrow \text{true}
\end{aligned}
$$

Following the same idea as above we match in the first case F with app : exp $\to$ (exp $\to$ exp) and return some object M_a (representing false). The arguments of app might mention the free parameter x, introduced by the case subject. Hence, before boxing each argument, it must be closed by abstracting over x: box $(\lambda x\!:\!\text{exp}.\, E_1\ x)$ and box $(\lambda x\!:\!\text{exp}.\, E_2\ x)$, both of type $\Box(\text{exp} \to \text{exp})$. The type of M_a is therefore $\Box(\text{exp} \to \text{exp}) \to \Box(\text{exp} \to \text{exp}) \to \Box\text{bool}$, and $M_a = \lambda E_1\!:\!\Box(\text{exp} \to \text{exp}).\, \lambda E_2\!:\!\Box(\text{exp} \to \text{exp}).\, \text{box false}$.

A very similar argument can be applied to determine the type of M_l, which is the match for lam : (exp $\to$ exp) $\to$ exp. x can occur free in the body E of the λ-expression, hence M_l will be passed the boxed object $\lambda x\!:\!\text{exp}.\, E$ which gives M_l the type $\Box(\text{exp} \to (\text{exp} \to \text{exp})) \to \Box\text{bool}$.

The parameter x might occur in the body of the case subject. Here again, as for the iterator, the matching object M_x is not expressed as a part of the case construct, the case object is rather a function expecting M_x as an argument. M_x must be of type $\Box$bool.

The identity test function is hence represented as follows.

$$\begin{aligned}\text{id-test} \;:\; & \Box(\text{exp} \to \text{exp}) \to \Box\text{bool} \\ = \;& \lambda E\!:\!\Box(\text{exp} \to \text{exp}).\, \text{case}\ \langle\Box\text{bool}\rangle\ E \\ & \quad \langle\, \text{app} \Rightarrow \lambda E_1\!:\!\Box(\text{exp} \to \text{exp}).\, \lambda E_2\!:\!\Box(\text{exp} \to \text{exp}).\, \text{box false} \\ & \quad \mid \text{lam} \Rightarrow \lambda E\!:\!\Box(\text{exp} \to \text{exp} \to \text{exp}).\, \text{box false}\rangle\ (\text{box true})\end{aligned}$$

To show how more than one case construct can be nested we develop briefly two functions to test if an expression from Example 2.3 is a β-redex or if it is an η-redex. To remind the reader β- and η-reduction are defined as follows.

$$\begin{aligned}&\beta\text{-reduction: } (\lambda x.\, E_1)\ E_2 \leadsto [E_2/x](E_1) \\ &\eta\text{-reduction: } \lambda x.\, (E\ x) \quad \leadsto E \quad \text{where } x \text{ does not occur free in } E\end{aligned}$$

$(\lambda x.\, E_1)\ E_2$ is called a β redex, $\lambda x.\, (E\ x)$ is called an η-redex if x does not occur free in E. These examples can be easily extended to the actual reduction functions.

Example 5.6 (β-redex test) The β-redex test function has type $\Box\text{exp} \to \Box\text{bool}$ and can informally be defined as follows.

$$\begin{aligned}\text{beta-test } F = \text{case } F \text{ of } & (\text{lam } E) \Rightarrow \text{false} \\ & \mid (\text{app } E_1\ E_2) \Rightarrow\ (\text{case } E_1 \text{ of } (\text{lam } E') \Rightarrow \text{true} \\ & \qquad\qquad\qquad\qquad \mid (\text{app } E_1'\ E_2') \Rightarrow \text{false})\end{aligned}$$

Its representation in our calculus is:

$$\begin{aligned}\text{beta-test} \;:\; & \Box\text{exp} \to \Box\text{bool} \\ = \;& \lambda F\!:\!\Box\text{exp}.\ \text{case}\ \langle\Box\text{bool}\rangle\ F \\ & \qquad \langle\, \text{lam} \Rightarrow \lambda E\!:\!\Box(\text{exp} \to \text{exp}).\, \text{box false} \\ & \qquad \mid \text{app} \Rightarrow \lambda E_1\!:\!\Box\text{exp}.\, \lambda E_2\!:\!\Box\text{exp}. \\ & \qquad\quad \text{case}\ \langle\Box\text{bool}\rangle\ E_1 \\ & \qquad\quad \langle\, \text{lam} \Rightarrow \lambda E'\!:\!\Box(\text{exp} \to \text{exp}).\, \text{box true} \\ & \qquad\quad \mid \text{app} \Rightarrow \lambda E_1'\!:\!\Box\text{exp}.\, \lambda E_2'\!:\!\Box\text{exp}.\, \text{box false}\rangle\rangle\end{aligned}$$

Example 5.7 (η-redex test) The function to decide if a given expression is a η-redex is more difficult to define. Clearly, it will have type $\Box\text{exp} \to$

$\Box$bool. The main difficulties arise because the decision cannot simply be made by considering the structure of the expression, but we must ensure the side condition for η-redices. This can be accomplished using the functions const (from Example 4.7) and id-test defined above.

$$
\begin{array}{ll}
\text{eta-test } F = \text{case } F \text{ of} & \\
\qquad (\text{lam } E) \Rightarrow \ \text{case } E \text{ of } \lambda x\!:\!\text{exp}.\, (\text{lam } \lambda y\!:\!\text{exp}.\, E'\ x\ y) \Rightarrow \text{false} \\
\qquad\qquad\qquad \mid \lambda x\!:\!\text{exp}.\, (\text{app } (E_1'\ x)\ (E_2'\ x)) \Rightarrow \\
\qquad\qquad\qquad\qquad (\text{and } (\text{const } E_1')\ (\text{id-test } E_2')) \\
\qquad\qquad\qquad \mid \lambda x\!:\!\text{exp}.\, x \Rightarrow \text{false} \\
\qquad \mid (\text{app } E_1\ E_2) \Rightarrow \text{false}
\end{array}
$$

Its representation in our calculus is:

$$
\begin{array}{l}
\text{eta-test} \ : \ \Box\text{exp} \to \Box\text{bool} \\
\qquad = \lambda F\!:\!\Box\text{exp}. \\
\qquad\quad \text{case } \langle \Box\text{bool} \rangle\ F \\
\qquad\quad \langle\, \text{lam} \Rightarrow \lambda E\!:\!\Box(\text{exp} \to \text{exp}). \\
\qquad\qquad \text{case } \langle \Box\text{bool} \rangle\ E \\
\qquad\qquad \langle\, \text{lam} \Rightarrow \lambda E'\!:\!\Box(\text{exp} \to \text{exp} \to \text{exp}).\, \text{box false} \\
\qquad\qquad \mid \text{app} \Rightarrow \lambda E_1'\!:\!\Box(\text{exp} \to \text{exp}).\, \lambda E_2'\!:\!\Box(\text{exp} \to \text{exp}). \\
\qquad\qquad\qquad (\text{and } (\text{const } E_1')\ (\text{id-test } E_2'))\rangle\ (\text{box false}) \\
\qquad\quad \mid \text{app} \Rightarrow \lambda E_1\!:\!\Box\text{exp}.\, \lambda E_2\!:\!\Box\text{exp}.\, (\text{box false})\rangle
\end{array}
$$

5.2 Formal Discussion

We begin now with the formal discussion of the case construct: Differently from iteration which traverses the entire structure of the subject, case only recurses down to the head constructor of the subject leaving possible arguments aside. The subject for selection is always of the form $\lambda x_1 : B_1. ..\lambda x_m : B_m. c\ M_1..M_n$ with a head constructor c of type B'. Operationally speaking, during the process of selection, the head constructor is replaced by an object M representing the selected case. At a first glance one might suspect that M's type is $B_1' \to .. \to B_n' \to A$ where the B_i''s are the argument types of c and A is the result type of the case. This is not powerful enough. Since case distinction requires its subject to be closed, no further case distinction could be performed over any of the objects $M_1..M_n$. To solve this dilemma we close each argument M_i by abstracting over each variable which might possibly occur free in it. It should be clear that all those variables can be determined because each M_i is a subobject of the case subject. This allows us to finally close the newly constructed object with a box. To make this more formal we define a generalized λ-abstraction which we call *abstraction closure*: $\lambda\{\Psi\}.\, M$ stands for a closed object where M is wrapped in λ-abstractions defined by Ψ

$$\lambda\{\cdot\}.\, M := M \qquad \lambda\{\Psi, x : B\}.\, M := \lambda\{\Psi\}.\, (\lambda x{:}B.\, M)$$

and similarly its type is defined as $\Pi\{\Psi\}.\, A$ even though no dependencies are involved, and the variable names are not used:

$$\Pi\{\cdot\}.\, A := A \qquad \Pi\{\Psi, x : B\}.\, A := \Pi\{\Psi\}.\, (B \to A)$$

Returning to our discussion we can now write box $(\lambda\{\Psi\}.\, M_i)$ for the abstracted and closed versions of M_i where Ψ is a context accounting for all free variables possibly occurring in M_i. It follows that this argument closing operation determines the type of M which we discuss next.

Example 5.5 raised the problem of assigning types to the arguments of the objects M_a and M_l which represent the computational meaning of the cases app and lam, respectively. Generalizing the idea proposed in this example leads to the notion of *case types*. As pointed out above, the general form of the canonical case subject is $\lambda\{\Psi\}.\, h\ M_1..M_n$ with a head constructor h of type B'. The type of the case subject is hence $\Pi\{\Psi\}.\, a$ for some atomic type a. Hence h can either be a constructor (defined in Σ) or a parameter (defined in Ψ) with target type a. Selecting a case for h means to select an object M_h. M_h must be a function, which expects as arguments box $(\lambda\{\Psi\}.\, M_1)$... box $(\lambda\{\Psi\}.\, M_n)$. Its type can hence be derived from the type of the case subject $B = \Pi\{\Psi\}.\, a$, the result type A, and the type of the constructor $h : B'$. We call the type of M_h *inner case type* and abbreviate it by $\mathcal{C}\ (\Pi\{\Psi\}.\, a, A, B')$:

Definition 5.8 (Inner case types)

$$\mathcal{C}\ ((\Pi\{\Psi\}.\, a), A, a') := \begin{cases} A \ \textit{if}\ a = a' \\ a' \ \textit{otherwise} \end{cases}$$

$$\mathcal{C}\ ((\Pi\{\Psi\}.\, a), A, (B_1 \to B_2)) := \Box(\Pi\{\Psi\}.\, B_1) \to \mathcal{C}\ ((\Pi\{\Psi\}.\, a), A, B_2)$$

Note, that for all examples so far $\tau(B) = \tau(B')$. Hence the *otherwise* case in the definition above does not apply for any of these examples. This changes for the next example.

Example 5.9 (Equality formulas in higher order logic) Consider a function which returns true if a higher-order formula is of the form $t_1 = t_2$, otherwise false. We call this function eq-test. The type of this function should be $\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \Box\mathrm{bool}$. Informally we would write:

$$\begin{aligned}\text{eq-test } F = \text{case } F \text{ of}&\\ &\lambda r\!:\!\mathrm{o} \to \mathrm{i}.\,\text{forall } (\lambda x\!:\!\mathrm{o}.\,F'\ r\ x) \Rightarrow \text{false}\\ &\lambda r\!:\!\mathrm{o} \to \mathrm{i}.\,\text{impl } (F_1'\ r)\ (F_2'\ r) \Rightarrow \text{false}\\ &\lambda r\!:\!\mathrm{o} \to \mathrm{i}.\,\text{eq } (t_1'\ r)\ (t_2'\ r) \Rightarrow \text{true}\end{aligned}$$

The straightforward representation of this function in our system is

$$\begin{aligned}\text{eq-test} &: \Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to (\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \mathrm{i}) \to \Box\mathrm{bool}\\ &= \lambda F\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}).\\ &\qquad \text{case } \langle \Box\mathrm{bool}\rangle\ F\\ &\qquad \langle\, \text{forall} \Rightarrow \lambda F'\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{i} \to \mathrm{o}).\,\text{box false}\\ &\qquad \mid \text{impl} \Rightarrow \lambda F_1'\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}).\,\lambda F_2'\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}).\\ &\qquad\qquad\qquad \text{box false}\\ &\qquad \mid \text{eq} \Rightarrow \lambda t_1'\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{i}).\,\lambda t_2'\!:\!\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{i}).\,\text{box true}\rangle\end{aligned}$$

The type of eq-test seems unnecessarily cluttered. This stems from the observation that the parameter r from the informal presentation can never occur in the head position of F. It is an immediate consequence of $\tau(\mathrm{o} \to \mathrm{i}) \neq \mathrm{o}$.

Hence one would expect eq-test's type to be $\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \Box\mathrm{bool}$, omitting the second argument type $\Box((\mathrm{o} \to \mathrm{i}) \to \mathrm{o}) \to \mathrm{i}$. Currently, our system does not treat this special case for the sake of simplicity of the meta theoretical discussion in Section 7. Thus a dummy argument must be supplied when executing eq-test.

The type of case $\langle A\rangle\ M\ \langle\Xi\rangle$ is called an *outer case type* $\mathcal{C}^*\ (B, A, B)$ where B is the type of M. $\mathcal{C}^*\ (B, A, B')$ is defined for some pure type B' as follows.

Definition 5.10 (Outer case type)

$$\mathcal{C}^*\ (B, A, a) := \mathcal{C}\ (B, A, a)$$

$$\mathcal{C}^*\ (B, A, (B_1 \to B_2)) := \mathcal{C}\ (B, A, B_1) \to \mathcal{C}^*\ (B, A, B_2)$$

The result of the selection process — i.e. the execution of the case construct — is an object which resembles the (canonical) subject of the case in structure, but the head constant is replaced by some *matched* object carrying the intended operational meaning of the selected branch. Even though the subject of case is closed before the selection process, we need to deal with embedded λ-abstractions introducing parameters. We can simply replace variables x of type B' by new variables x' of type $\mathcal{C}\ (B, A, B')$, where B is the type of the case subject.

Definition 5.11 (Match)

$$\textit{Matches: } \Xi ::= \cdot \mid (\Xi \mid c \Rightarrow M)$$

The domain of a match is a sub-signature $\mathcal{S}(\Sigma; \tau(B))$ containing all constructors whose target type equals $\tau(B)$. The form of case follows naturally: We extend the notion of objects by

$$M ::= \ldots \mid \text{case } \langle A \rangle\ M\ \langle \Xi \rangle$$

and extend the typing rules for case. To do so we must introduce a new typing judgment for matches Ξ: $\Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A \rangle(\Sigma)$. Ξ is well-typed if it provides an object of inner case type for every constant in some signature Σ.

Definition 5.12 (Typing judgment for case) *extending definition 3.1*

$$\frac{\Delta; \Gamma \vdash M : \Box B \qquad \Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A \rangle(\mathcal{S}(\Sigma; \tau(B)))}{\Delta; \Gamma \vdash \textit{case}\ \langle A \rangle\ M\ \langle \Xi \rangle : \mathcal{C}^*\ (B, A, B)}\ \mathsf{TpCase}$$

$$\frac{}{\Delta; \Gamma \vdash \cdot : \langle B \Rightarrow A \rangle(\cdot)}\ \mathsf{TmBase}$$

$$\frac{\Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A \rangle(\Sigma) \qquad \Delta; \Gamma \vdash M : \mathcal{C}\ (B, A, B')}{\Delta; \Gamma \vdash (\Xi \mid c \Rightarrow M) : \langle B \Rightarrow A \rangle(\Sigma, c : B')}\ \mathsf{TmInd}$$

To summarize definition by cases we return to Example 5.5 (id-test). As in the iteration case, it is necessary to first derive the type of the function, because the subject of case must be closed. id-test has type $\Box(\text{exp} \to \text{exp}) \to \Box\text{bool}$.

The second step is to examine the argument type $B = (\text{exp} \to \text{exp})$ and the signature Σ for possible constructors and parameters of this type. For id-test we find only three candidates: lam, app, and x (the newly introduced parameter). After determining their types the match objects M_l, M_a, and M_x must be defined.

Having done this, a match must be constructed, representing only the constructors from the signature Σ and the according case objects ($\Xi = \text{lam} \Rightarrow M_l, \text{app} \Rightarrow M_a$). The case construct then results in a function which must be applied to the object M_x.

The operational semantics of case is defined by one rule using the concept of *selection*. Because the subject of case is closed we follow the example of elimination and define selection along its canonical form V (of type B). Even though the subject of case is closed before the selection process, we need to deal with embedded λ-abstractions introducing parameters. We can simply replace variables x of type B' by new variables x' of type $\mathcal{C}\ (B, A, B')$, where B is the type of the case subject. The definition of a match is extended accordingly.

Definition 5.13 (Match)

$$\textit{Match:}\ \Xi ::= \ldots \mid (\Xi \mid x \Rightarrow x')$$

The selection process then transforms V of type B into $\{B \Rightarrow A; \Xi; \cdot\}(V)$ of type $\mathcal{C}^*\ (B, A, B)$. That this transformation is well-defined is a result we show later in this paper. The selection process is formally defined by the following rules.

Definition 5.14 (Selection)

$$\{B \Rightarrow A; \Xi; \Psi\}(c) \quad = \Xi(c) \qquad (\mathsf{SeConst})$$

$$\{B \Rightarrow A; \Xi; \Psi\}(x) \quad = \Xi(x) \qquad (\mathsf{SeVar})$$

$$\{B \Rightarrow A; \Xi; \Psi\}(\lambda x{:}B'.\, V) =$$

$$\lambda u{:}\mathcal{C}\ (B, A, B').\, \{B \Rightarrow A; \Xi | x \Rightarrow u; (\Psi, x : B')\}(V) \qquad (\mathsf{SeLam})$$

$$\{B \Rightarrow A; \Xi; \Psi\}(V_1\ V_2) \quad = \{B \Rightarrow A; \Xi; \Psi\}(V_1)\ (\mathit{box}\ \lambda\{\Psi\}.\, V_2) \qquad (\mathsf{SeApp})$$

An arbitrary canonical form of a case subject of type B has always the form $\lambda\{\Psi\}.\, c\ M_1..M_n$. Performing selection means to first traverse all λ-abstractions, and introducing new variables for each parameter. This is done by rule SeLam. While traversing the body of the canonical form, each argument M_i must be closed under Ψ and boxed which is expressed by rule SeApp. Eventually the head constructor c is reached. If c is a constructor/parameter then SeVar/SeConst replaces it by the corresponding object from match Ξ.

The selection process is triggered by an additional evaluation rule, which defines the operational semantics of case.

Definition 5.15 (Evaluation judgment) *extending Definition 3.3:*

$$\frac{\begin{array}{c}\Psi \vdash M \hookrightarrow \mathit{box}\, M' : \Box B \\ \cdot \vdash M' \Uparrow V' : B\end{array} \qquad \Psi \vdash \{B \Rightarrow A; \Xi; \cdot\}(V') \hookrightarrow V : \mathcal{C}^* \ (B, A, B)}{\Psi \vdash \mathit{case}\ \langle A \rangle\ M\ \langle \Xi \rangle \hookrightarrow V : \mathcal{C}^* \ (B, A, B)}\ \textsf{EvCase}$$

The reader can now convince himself that the operational semantics yields the expected results on the examples of this section. This concludes the presentation of the modal λ-calculus. In the next sections, we discuss its metatheoretical properties.

6 Preliminary results

In the remainder of the paper we seek to prove that the modal λ-calculus is a conservative extension over the simply-typed λ-calculus from Section 2. A milestone on the way towards this result is the canonical form theorem which we present in the next section. It guarantees that every object of pure type possesses a canonical form. As a corollary of the canonical form theorem we obtain a type preservation result which guarantees that types are preserved under evaluation.

In the remainder of this paper we will need some more basic technical notions and properties which we are presenting in this section. Due to the basic character, a lot of the forthcoming lemmas are clear and their proofs do not require more than easy inductive arguments. If appropriate we omit the proofs. All other lemmas in this and in the following sections require lengthy proofs. For the sake of brevity, we give only a brief summary of each proof. Full details of the proofs may be found in [DPS97].

6.1 Context

In Section 3 we described the distinction between the parametric function space $A_1 \to A_2$ and the primitive recursive function space $A_1 \Rightarrow A_2$ which made a refinement of context Ψ from Section 2 necessary: We introduced the *modal* context Δ, whose variables range over closed objects and the *arbitrary* context Γ whose variables range over potentially open objects.

In the following discussion it will be necessary to reason about contexts. The arguments will involve extensions of contexts which we write as $\Gamma' \geq \Gamma$ and which are defined in the usual way. In the case of a modal/non-modal context pair $\Delta; \Gamma$, as it is used in the typing judgment, we implicitly stipulate that we only consider extensions Δ' of Δ which yield a valid context $\Delta'; \Gamma$. A similar remark holds for extensions of Γ.

Closely related to contexts are substitutions which we introduce after discussing the typing relation of our system.

6.2 Typing

One basic property which is needed in the proof of Lemma 7.14 in the next section is the admissibility of weakening for the typing relation. The following lemma has two parts, the first part discusses weakening in the modal context, the second weakening in the regular context. We omit the easy proof by induction.

Lemma 6.1 (Weakening)

(1) If $\Delta; \Gamma \vdash M : A$ *and* $\Delta' \geq \Delta$ *then* $\Delta'; \Gamma \vdash M : A$
(2) If $\Delta; \Gamma \vdash M : A$ *and* $\Gamma' \geq \Gamma$ *then* $\Delta; \Gamma' \vdash M : A$

Besides weakening we require two substitution lemmas. One which allows substituting a closed well-typed object for a modal variable, and another which allows substituting an arbitrary well-typed object for a non-modal variable. We again omit the easy inductive proofs.

Lemma 6.2 (Modal substitution lemma)

If $\Delta, y : A_1; \Gamma \vdash M : A_2$ *and* $\Delta; \cdot \vdash M' : A_1$ *then* $\Delta; \Gamma \vdash [M'/y](M) : A_2$

Lemma 6.3 (Regular substitution lemma)

If $\Delta; \Gamma, y : A_1 \vdash M : A_2$ *and* $\Delta; \Gamma \vdash M' : A_1$ *then* $\Delta; \Gamma \vdash [M'/y](M) : A_2$

Contexts and substitutions are closely related. A substitution is defined as $\varrho ::= \cdot \mid \varrho, M/x$. Due to the presence of two contexts we carefully distinguish between a modal substitution θ which substitutes closed objects for variables defined in a context Δ and ϱ which substitutes arbitrary objects for variables defined in a context Γ. We write $\theta; \varrho$ for such a pair of (necessarily disjoint) substitutions. Being disjoint means, that θ and ϱ do not have any variable names in common in their domains. This is guaranteed, because the contexts $\Delta; \Gamma$ cannot declare the same variable name twice.

In our system substitutions are only applied to well-typed objects. Moreover a substitution must substitute something for every free variable in the object. We make this intuition about well-typed substitutions more precise by introducing a typing judgment $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ for substitutions. $\theta; \varrho$ can be applied to objects which are well-typed in context $\Delta; \Gamma$. The range of the substitution $\theta; \varrho$ are objects which might depend on free variables from $\Delta'; \Gamma'$.

Definition 6.4 (Typing substitutions)

$$\frac{}{\Delta'; \Gamma' \vdash (\cdot; \cdot) \ : \ (\cdot; \cdot)}\ \mathsf{TSBase}$$

$$\frac{\Delta'; \cdot \vdash M : A \qquad \Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)}{\Delta'; \Gamma' \vdash (\theta, M/x; \varrho) \ : \ (\Delta, x : A; \Gamma)}\ \mathsf{TSMod}$$

$$\frac{\Delta'; \Gamma' \vdash M : A \qquad \Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)}{\Delta'; \Gamma' \vdash (\theta; \varrho, M/x) \ : \ (\Delta; \Gamma, x : A)}\ \mathsf{TSReg}$$

Throughout this paper we apply a substitution $(\theta; \varrho)$ satisfying $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ only to well-typed objects M, well-typed term replacements Ω, and well-typed matches Ξ. The application of a substitution $\theta; \varrho$ is defined as follows:

Definition 6.5 (Substitution application) *Let $\theta; \varrho$ a substitution.*

$$[\theta; \varrho](x) = \begin{cases} M \ \textit{if}\ \theta(x) = M \\ M \ \textit{if}\ \varrho(x) = M \end{cases} \tag{SBVar}$$

$$[\theta; \varrho](c) = c \tag{SBConst}$$

$$[\theta; \varrho](\lambda x : A.\, M) = \lambda x : A.\, [\theta; \varrho, x/x](M) \tag{SBLam}$$

$$[\theta; \varrho](M_1\ M_2) = [\theta; \varrho](M_1)\ [\theta; \varrho](M_2) \tag{SBApp}$$

$$[\theta;\varrho](\langle M_1, M_2\rangle) = \langle[\theta;\varrho](M_1), [\theta;\varrho](M_2)\rangle \qquad (\mathsf{SBPair})$$
$$[\theta;\varrho](\mathit{fst}\ M) = \mathit{fst}\ [\theta;\varrho](M) \qquad (\mathsf{SBFst})$$
$$[\theta;\varrho](\mathit{snd}\ M) = \mathit{snd}\ [\theta;\varrho](M) \qquad (\mathsf{SBSnd})$$
$$[\theta;\varrho](\mathit{box}\ M) = \mathit{box}\ [\theta;\cdot](M) \qquad (\mathsf{SBBox})$$
$$[\theta;\varrho](\mathit{let\ box}\ x = M_1\ \mathit{in}\ M_2) = \mathit{let\ box}\ x = [\theta;\varrho](M_1)\ \mathit{in}\ [\theta, x/x;\varrho](M_2) \qquad (\mathsf{SBLet})$$
$$[\theta;\varrho](\mathit{case}\ \langle A\rangle\ M\ \langle\Xi\rangle) = \mathit{case}\ \langle A\rangle\ [\theta;\varrho](M)\ \langle[\theta;\varrho](\Xi)\rangle \qquad (\mathsf{SBCase})$$
$$[\theta;\varrho](\mathit{it}\ \langle\omega\rangle\ M\ \langle\Omega\rangle) = \mathit{it}\ \langle\omega\rangle\ [\theta;\varrho](M)\ \langle[\theta;\varrho](\Omega)\rangle \qquad (\mathsf{SBIt})$$

Substitution on replacements Ω is defined as:

$$[\theta;\varrho](\cdot) = \cdot \qquad (\mathsf{SBOmegaEmpty})$$
$$[\theta;\varrho](\Omega|c \mapsto M) = [\theta;\varrho](\Omega)\ |\ (c \mapsto [\theta;\varrho](M)) \qquad (\mathsf{SBOmega})$$

Substitution on matches Ξ is defined as:

$$[\theta;\varrho](\cdot) = \cdot \qquad (\mathsf{SBXiEmpty})$$
$$[\theta;\varrho](\Xi\ |\ c \Rightarrow M) = [\theta;\varrho](\Xi)\ |\ (c \Rightarrow [\theta;\varrho](M)) \qquad (\mathsf{SBXi})$$

The rule SBVar is well-defined because every variable subject to substitution is uniquely defined either in θ or in ϱ. SBBox is non-standard. Since a boxed term is closed it can only contain variables representing closed objects and no variables representing arbitrary objects. This is easily verified by inversion of the TpBox rule because we assume the subject of substitution always to be well-typed. This means, that ϱ will not be used during the substitution process and can hence be discarded.

We write id_Γ for the identity substitution. The identity substitution mapping the context $\Delta;\Gamma$ to itself has hence the form: $\Delta;\Gamma \vdash (\mathrm{id}_\Delta;\mathrm{id}_\Gamma)\ :\ (\Delta;\Gamma)$.

Two different notions of substitution have been used in the presentation of our system so far: One is the substitution $(\theta;\varrho)$, the other is the substitution as used for example in the evaluation rules EvApp and EvLet: $[M_1/x](M_2)$. They interact in the following way:

Lemma 6.6 (Property of substitutions)

(1) $[M'/x]([\theta;\varrho, x/x](M)) = [\theta;\varrho, M'/x](M)$

(2) $[M'/x]([\theta, x/x; \varrho](M)) = [\theta, M'/x; \varrho](M)$

Weakening is also admissible for the typing judgment of substitutions.

Lemma 6.7 (Weakening for substitutions)

(1) If $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ *and* $\Delta'' \geq \Delta'$ *then* $\Delta''; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$
(2) If $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ *and* $\Gamma'' \geq \Gamma'$ *then* $\Delta'; \Gamma'' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$

By induction we can prove that restricting a well-typed substitution $\theta; \varrho$ to $\theta; \cdot$ is also well-typed. If the domain of $\theta; \varrho$ is $\Delta; \Gamma$, the domain of the restricted substitution is clearly $\Delta; \cdot$.

Lemma 6.8 (Modal substitution restriction)

If $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ *then* $\Delta'; \cdot \vdash (\theta; \cdot) \ : \ (\Delta; \cdot)$

Proof: by induction on the derivation of $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$ (see [DPS97, Lemma 6.19]) $\square$

These preparatory results lead to a general substitution lemma for the typing judgment. If an object M is well-typed in a context for which a substitution is well-defined then its application to M yields an object of the same type as M in the context of the substitution. Since objects are also defined in terms of term replacements and matches we must extend the result to both constructs.

Lemma 6.9 (Substitution lemma for typing relation)

Let $\Delta'; \Gamma' \vdash (\theta; \varrho) \ : \ (\Delta; \Gamma)$*, then the following holds:*

(1) If $\Delta; \Gamma \vdash M : A$ *then* $\Delta'; \Gamma' \vdash [\theta; \varrho](M) : A$
(2) If $\Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A \rangle(\Sigma')$ *then* $\Delta'; \Gamma' \vdash [\theta; \varrho](\Xi) : \langle B \Rightarrow A \rangle(\Sigma')$
(3) If $\Delta; \Gamma \vdash \Omega : \langle \omega \rangle(\Sigma')$ *then* $\Delta'; \Gamma' \vdash [\theta; \varrho](\Omega) : \langle \omega \rangle(\Sigma')$

Proof: by mutual induction on the derivations of $\Delta; \Gamma \vdash M : A$, $\Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A \rangle(\Sigma')$ and $\Delta; \Gamma \vdash \Omega : \langle \omega \rangle(\Sigma')$, using Lemmas 6.1, 6.7, and 6.8 (see [DPS97, Lemma 6.21]) $\square$

As corollary we can apply the substitution lemma to the identity substitution and obtain (by a short inductive argument) the trivial result that if $\Delta; \Gamma \vdash M : A$ then $[\mathrm{id}_\Delta; \mathrm{id}_\Gamma](M) = M$.

For atomic and canonical form judgments there is also a weakening result. The proof is a straightforward mutual induction on the derivations of atomic and canonical forms, and we omit it here.

Lemma 6.10 (Weakening for atomic and canonical forms)

(1) If $\Psi \vdash V \downarrow B$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash V \downarrow B$
(2) If $\Psi \vdash V \Uparrow B$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash V \Uparrow B$

A slightly more complicated property of atomic and canonical forms is that the type of an object can be directly inferred from the judgment.

Lemma 6.11 (Typing of atomic and canonical forms)

(1) If $\Psi \vdash V \downarrow B$ *then* $\cdot; \Psi \vdash V : B$
(2) If $\Psi \vdash V \Uparrow B$ *then* $\cdot; \Psi \vdash V : B$

Proof: by mutual induction on the derivations of $\Psi \vdash V \downarrow B$ and $\Psi \vdash V \Uparrow B$ (see [DPS97, Lemma 6.25]) □

Evaluation derivations can also be weakened. The omitted proof proceeds by induction on the evaluation derivation.

Lemma 6.12 (Weakening for evaluation)

If $\Psi \vdash M \hookrightarrow V : A$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash M \hookrightarrow V : A$

6.5 Subordination of types

The subordination relation accounts for all dependencies which are introduced by the signature or by the subject type of iteration or case. In the remainder of this section we characterize and discuss a few major properties of type subordination which will prove very useful when we tackle the proof of the canonical form theorem.

There is a close relationship between source types of a sub-signature and the subordination relation. This relationship can be characterized by the following observation: Every source type of the sub-signature is trivially subordinate to the target type of the constructor. Furthermore, the weak subordination relation is transitive: If a type a_1 is subordinate to a type a_2 and a_2 is weakly subordinate to a type a_3, then a_1 is automatically subordinate to a_3. In this case the result follows trivially from the Definition 4.23 of subordination.

Lemma 6.13 (Properties of subordination) *Let $c : C \in \Sigma$, B a pure type.*

(1) If $a \in Source(C)$ then $a \blacktriangleleft_{\Sigma;B} \tau(C)$
(2) If $a_1 \blacktriangleleft_{\Sigma;B} a_2$ and $a_2 \trianglelefteq_{\Sigma;B} a_3$ then $a_1 \blacktriangleleft_{\Sigma;B} a_3$

Proof: direct (see [DPS97, Lemma 6.30]) □

Since parameters are variable names which are represented in a context, we must extend the notion of subordination to contexts: For all parameter types B' if the target type $\tau(B')$ is subordinate to a, all source types of B' are also subordinate to a.

Definition 6.14 (Subordination on contexts) *Let a be an atomic type.*

$$\cdot \blacktriangleleft_{\Sigma;B} a$$

$$\begin{array}{l} \Psi, x : B' \blacktriangleleft_{\Sigma;B} a \\ \qquad \textit{iff } \Psi \blacktriangleleft_{\Sigma;B} a \textit{ and } \textit{if } \tau(B') \trianglelefteq_{\Sigma;B} a \\ \qquad\qquad\qquad\qquad \textit{then for all } \ y \in Source(B') : y \blacktriangleleft_{\Sigma;B} a \end{array}$$

It will become clear during the proof of the canonical form theorem, how context subordination is used. If a variable x of type B' is defined in a context Ψ and $\Psi \blacktriangleleft_{\Sigma;B} \tau(B)$ then all source types of type B' are automatically subordinate to the goal type of B.

Lemma 6.15 (Properties of context subordination) *Let B a pure type.*

If $\Psi = \Psi_1, x : B', \Psi_2$ and $\Psi \blacktriangleleft_{\Sigma;B} \tau(B)$ then $\tau(B') \trianglelefteq_B \tau(B)$ implies that for all $y \in Source(B')$: $y \blacktriangleleft_{\Sigma;B} \tau(B)$

Proof: direct (see [DPS97, Lemma 6.32]) □

As we have motivated earlier, only the top-level parameters, defined by the type of the iteration or case subject B, are relevant for the subordination relation. We inductively define the *set of parameter types* introduced by B as follows.

Definition 6.16 (Set of parameter types)

$$\mathcal{P}(a) = \{\}$$

$$\mathcal{P}(B_1 \to B_2) = \{B_1\} \cup \mathcal{P}(B_2)$$

It can be shown, that if B' is such a parameter type then all source types of B' must be also source types of B. This is clear, because every parameter type corresponds to a source type of B and B must be a function type. Every source type of B' is hence a source type of B. We omit the proof.

Lemma 6.17 (Subset property of P)

For all $B' \in \mathcal{P}(B)$ *:* $\mathit{Source}(B') \subseteq \mathit{Source}(B)$

If a parameter type B' of a type B is given, and a is a type which is immediately subordinate to the target type of B', then we have $a \lhd_B \tau(B')$, because every source type of B' is also a source type of B.

Lemma 6.18 (Property of dynamic typing)

If $B' \in \mathcal{P}(B)$ *then* $a <_{B'} \tau(B')$ *implies* $a \lhd_B \tau(B')$

Proof: by induction on B (see [DPS97, Lemma 6.34]) □

While iteration traverses a subject of type B it may encounter constants of type C whose target type does not occur in the equivalence class $\mathcal{I}(\Sigma; B)$. We have seen this in Example 4.8 where nat $\notin \mathcal{I}(\Sigma; \mathrm{db})$ and in Example 4.5 where i $\notin \mathcal{I}(\Sigma; \mathrm{o})$. For such a constant c, $\tau(C)$ is always subordinate to $\tau(B)$ as we will see in Lemma 7.32 (Auxiliary lemma for iteration), but never vice versa. Since the elimination process $\langle\omega; \Omega\rangle(M)$ must return a well-typed object, we must require that ω maps c's constructor type C to C. Thus, more formally, we must show that if $\tau(C) \notin \mathcal{I}(\Sigma; B)$ and $\tau(C) \trianglelefteq_B \tau(B)$ then $\langle\omega\rangle(C) = C$. We split this proof into three lemmas.

Lemma 6.19 (Properties of subordination)

If $\tau(C) \notin \mathcal{I}(\Sigma; B)$ *and* $\tau(C) \trianglelefteq_{\Sigma;B} \tau(B)$ *then* $\tau(B) \not\trianglelefteq_{\Sigma;B} \tau(C)$

Proof: by contradiction (see [DPS97, Lemma 6.37]) □

If the target type of B is not subordinate to the target type of C, then none of C's source types can be a member of the equivalence class. If it were then it would also be subordinate to the target type of B, violating our assumption.

Lemma 6.20 (Independence)

If $\tau(B) \not\prec_{\Sigma;B} \tau(C)$ *then* $Source(C) \cap \mathcal{I}(\Sigma; B) = \emptyset$

Proof: by contradiction (see [DPS97, Lemma 6.35]) □

The third lemma ensures that the type replacement acts as identity on all types C whose source types are not in its domain.

Lemma 6.21 (Properties of type replacement) *Let* $c : C \in \Sigma$, α *arbitrary and* $\vdash \omega : \alpha$

If $Source(C) \cap \alpha = \emptyset$ *and* $\tau(C) \notin \alpha$ *then* $\langle\omega\rangle(C) = C$

Proof: by induction on C (see [DPS97, Lemma 6.36]) □

This concludes the section of the basic preliminary results. In the next section we address the problem of the existence of canonical forms for well-typed objects in our calculus.

7 Canonical form theorem

The aim of this section is to prove the canonical form property of the modal λ-calculus. The main result is that every object of pure type in a pure context possesses a canonical form. In our notation this property is expressed as: if $\cdot;\Psi \vdash M : B$ then $\Psi \vdash M \Uparrow V : B$. This result implies the conservative extension property of our system which we will show in Section 9. We prove this by Tait's method, often called an argument by logical relations or reducibility candidates. In such an argument we construct an interpretation of types as a relation between objects, in our case a unary relation P. The proof using logical relation proceeds then in two steps. First, we show that each member of the logical relation evaluates to a canonical form. Second, we prove that each well-typed object must satisfy P and hence be a member of the logical relation.

Before we go into details of the logical relation in Section 7.2, we derive in Section 7.1 some lemmas which are essential for the argument, and prove the first step. The second step of the argument is presented in the following four sections: Section 7.3 establishes some basic results, Section 7.4 introduces some more logical relations — defined on substitutions — and eventually we motivate and prove an auxiliary lemma for iteration (Section 7.5) and an auxiliary lemma for case (Section 7.6). Finally, Section 7.7 discusses how to assemble all these results to a proof of the canonical form theorem.

7.1 Basic properties

Some of the following proofs rely on the fact, that canonical and atomic forms evaluate to themselves. This fact, even though it might seem trivial, requires a mutual inductive argument: To prove that atomic and canonical forms evaluate to themselves we must generalize the property in the following way:

Lemma 7.1 (Self evaluation)

(1) If $\Psi \vdash M \hookrightarrow V : B$ *and* $\Psi \vdash V \Uparrow B$ *then* $\Psi \vdash M \Uparrow V : B$
(2) If $\Psi \vdash V \Uparrow B$ *then* $\Psi \vdash V \hookrightarrow V : B$
(3) If $\Psi \vdash V \downarrow B$ *then* $\Psi \vdash V \hookrightarrow V : B$

Proof: by mutual induction on the derivation of $\Psi \vdash V \Uparrow B$, $\Psi \vdash V \Uparrow B$, and $\Psi \vdash V \downarrow B$, using Lemmas 6.11 and 6.12 (see [DPS97, Lemma 7.1]) $\square$

Another result which seems intuitively clear but must be proven is the following: by definition objects evaluate to other objects under the judgment $\Psi \vdash M \Uparrow V : B$. Since this is the judgment for canonical evaluation, we expect V to be canonical. Contrary to the intuition, the proof is not straightforward since the notion of conversion to canonical forms depends on the evaluation judgment. It is also not very sensible to try to prove that for every object M, $\Psi \vdash M \hookrightarrow V : A$ implies that V is a canonical form. For example, consider the signature from example 2.3: It is easy to see that

$$\cdot \vdash \lambda x\!:\!\text{exp}.\, (\lambda y\!:\!\text{exp}.\, y)\ \text{z} \hookrightarrow \lambda x\!:\!\text{exp}.\, (\lambda y\!:\!\text{exp}.\, y)\ \text{z} : \text{exp} \to \text{exp}$$

but it is also clear that $\lambda x : \text{exp}.\, (\lambda y\!:\!\text{exp}.\, y)\ \text{z}$ is not canonical, because the body of the λ-expression can be β-reduced. However, it holds when restricted to atomic types.

Lemma 7.2 (Property of evaluation results)

(1) If $\Psi \vdash M \Uparrow V : B$ *then* $\Psi \vdash V \Uparrow B$
(2) If $\Psi \vdash M \hookrightarrow V : a$ *then* $\Psi \vdash V \downarrow a$

Proof: by mutual induction on the derivations of $\Psi \vdash M \Uparrow V : B$ and $\Psi \vdash M \hookrightarrow V : B$ (see [DPS97, Lemma 7.2]) $\square$

The constant app from Example 2.3 is not a canonical form either. Canonical forms are objects in η-long β-normal form. app can be easily η-expanded to $\lambda x : \text{exp}.\, \lambda y\!:\!\text{exp}.\, \text{app}\ x\ y$. According to Lemma 7.2 (2) the result of an evaluation is canonical only if it is an object of atomic type. Nothing is said about functions. In general it can be shown that all objects M evaluating to an atomic form, possess a canonical form V'.

Lemma 7.3 (Canonical evaluation)

If $\Psi \vdash M \hookrightarrow V : B$ *and* $\Psi \vdash V \downarrow B$ *then* $\Psi \vdash M \Uparrow V' : B$ *for a* V'

Proof: by induction on B, using Lemmas 6.12 and 7.2 (see [DPS97, Lemma 7.3]) $\square$

7.2 Logical relation

Due to the lazy character of the modal λ-calculus, the interpretation of a type A is twofold: On the one hand 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 which must be an element of $|A|$. For the first we write $\Psi \vdash M \in [\![A]\!]$, for the second $\Psi \vdash V \in |A|$.

Definition 7.4 (Logical relation)

$\Psi \vdash M \in [\![A]\!] :\Leftrightarrow \cdot; \Psi \vdash M : A$ *and* $\Psi \vdash M \hookrightarrow V : A$ *and* $\Psi \vdash V \in |A|$
$\Psi \vdash V \in |A| :\Leftrightarrow$
- ***Case:*** $A = a$ *and* $\Psi \vdash V \Uparrow a$
- ***Case:*** $A = A_1 \rightarrow A_2$ *and*
  - ***either:*** $V = \lambda x : A_1. M$ *and for all* $\Psi' \geq \Psi$*:* $\Psi' \vdash V' \in |A_1| \Rightarrow \Psi' \vdash [V'/x](M) \in [\![A_2]\!]$
  - ***or:*** $\Psi \vdash V \downarrow A_1 \rightarrow A_2$ *and for all* $\Psi' \geq \Psi$*:* $\Psi' \vdash V' \Uparrow A_1 \Rightarrow \Psi' \vdash V\ V' \in |A_2|$
- ***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:*** $A = \Box A'$*:* $V =$ *box* M *and* $\cdot \vdash M \in [\![A']\!]$

The first logical relation requires its elements to be well-typed, a property which will be used in Lemma 7.36. $\Psi \vdash M \in [\![A]\!]$ must imply that M has type A in $\cdot; \Psi$. In Lemma 7.16 we show that this property propagates to the logical relation of values.

Objects were defined in terms of term replacements and matches (see Section 4, Section 5). Later in this section we need to show that every object defined in a term replacement or match is a member of a logical relation. To make our presentation of this circumstance cleaner and easier to understand we introduce the notion of logical relations for term replacements. A term replacement is an element of the logical relation defined by a signature Σ and a context representing parameters $\hat{\Psi}$, if every object associated with each parameter or constructor satisfies the appropriate logical relation. This relation is determined by the resulting type of applying the type replacement ω (defined by the iterator object) to the parameter or constructor type.

Definition 7.5 (Logical relation for term replacements) $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma; \hat{\Psi})]\!] :\Leftrightarrow$

Case: *If* $\hat{\Psi} = \cdot$ *and* $\Sigma = \cdot$ *then* $\Omega = \cdot$
Case: *If* $\hat{\Psi} = \cdot$ *and* $\Sigma = \Sigma', c : B$ *then* $\Omega = \Omega' \mid c \mapsto M$ *and* $\Psi \vdash M \in [\![\langle\omega\rangle(B)]\!]$ *and* $\Psi + \tilde{\Psi} \vdash \Omega' \in [\![\langle\omega\rangle(\Sigma'; \cdot)]\!]$
Case: *If* $\hat{\Psi} = \hat{\Psi}', x : B$ *then* $\Omega = \Omega' \mid x \mapsto u$ *and* $\tilde{\Psi} \vdash u \in [\![\langle\omega\rangle(B)]\!]$ *and* $\Psi + \tilde{\Psi} \vdash \Omega' \in [\![\langle\omega\rangle(\Sigma; \hat{\Psi}')]\!]$

The context defined for the logical relation of term replacements is split into two parts $\Psi + \tilde{\Psi}$. Ψ represents the context of variables which might occur free in the objects associated with constructors (note: *not* parameters), and $\tilde{\Psi}$ stands for the context of newly defined variables which rename the original parameters. We must keep both contexts separated, because to prove Lemma 7.35 and Lemma 7.32 we require a substitution, which acts as the identity on all variables defined in Ψ, but not necessarily on $\tilde{\Psi}$.

Every object in the logical relation $[\![A]\!]$ is well-typed by definition. This property propagates to term replacements.

Lemma 7.6 (Type preservation for term replacements) *If* $\Psi + \cdot \vdash \Omega \in [\![\langle\omega\rangle(\Sigma; \cdot)]\!]$ *then* $\cdot; \Psi \vdash \Omega : \langle\omega\rangle(\Sigma)$

Proof: by induction on Σ (see [DPS97, Lemma 7.6]) □

Similarly we define the logical relation for matches. A match is an element of the logical relation defined by a signature Σ and a context representing parameters $\hat{\Psi}$, if every object associated with each parameter/constructor satisfies the appropriate logical relation. This relation is determined by the inner case type of the parameter/constructor type. For the same reasons as for the term replacement we define the logical relation using two contexts: Ψ and $\tilde{\Psi}$.

Definition 7.7 (Logical relation for matches)
$\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi})]\!] :\Leftrightarrow$

Case: *If* $\hat{\Psi} = \cdot$ *and* $\Sigma = \cdot$ *then* $\Xi = \cdot$
Case: *If* $\hat{\Psi} = \cdot$ *and* $\Sigma = \Sigma', c : B'$ *then* $\Xi = \Xi' \mid c \Rightarrow M$ *and* $\Psi \vdash M \in [\![\mathcal{C}\ (B, A, B')]\!]$ *and* $\Psi + \tilde{\Psi} \vdash \Xi' \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \cdot)]\!]$
Case: *If* $\hat{\Psi} = \hat{\Psi}', x : B'$ *then* $\Xi = \Xi' \mid x \Rightarrow u$ *and* $\tilde{\Psi} \vdash u \in [\![\mathcal{C}\ (B, A, B')]\!]$ *and* $\Psi + \tilde{\Psi} \vdash \Xi' \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi}')]\!]$

Every object in the logical relation $[\![A]\!]$ is well-typed and so is every term replacement. As one might expect, this property can also be shown for matches.

Lemma 7.8 (Type preservation for matches)

If $\Psi + \cdot \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \cdot)]\!]$ *then* $\cdot; \Psi \vdash \Xi : \langle B \Rightarrow A\rangle(\Sigma)$

Proof: by induction on Σ (see [DPS97, Lemma 7.8]) □

The next few lemmas show some useful properties implied by the logical relations, all necessary to eventually prove the canonical form theorem. The first lemma is a standard weakening lemma for logical relations:

Lemma 7.9 (Weakening for logical relations)

(1) If $\Psi \vdash M \in [\![A]\!]$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash M \in [\![A]\!]$
(2) If $\Psi \vdash V \in |A|$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash V \in |A|$

Proof: by mutual induction on A, using Lemmas 6.1, 6.10, and 6.12 (see [DPS97, Lemma 7.9]) □

We also need a weakening result for the logical relation of term replacements. For our purposes it is enough to prove it with respect to $\tilde{\Psi}$.

Lemma 7.10 (Weakening for logical relations for replacement)

If $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma; \hat{\Psi})]\!]$ *and* $\tilde{\Psi}' \geq \tilde{\Psi}$ *then* $\Psi + \tilde{\Psi}' \vdash \Omega \in [\![\langle\omega\rangle(\Sigma; \hat{\Psi})]\!]$

Proof: by induction on $\Sigma, \hat{\Psi}$, using Lemma 7.9 (see [DPS97, Lemma 7.10])□

The logical relation for matches was defined analogously to the logical relation of term replacements. As expected the formulation of the weakening property is also analogous.

Lemma 7.11 (Weakening for logical relations for matches)

If $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi})]\!]$ *and* $\tilde{\Psi}' \geq \tilde{\Psi}$
then $\Psi + \tilde{\Psi}' \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi})]\!]$

Proof: by induction on $\Sigma, \hat{\Psi}$, using Lemma 7.9 (see [DPS97, Lemma 7.11])□

The logical relations for term replacements and matches play an important role when we discuss the elimination and selection process, respectively. Recall from Definition 4.29 that the elimination process traverses the structure of the iteration subject. Eventually constants or parameters will be encountered, and replaced by a term replacement Ω. In the proof of Lemma 7.32 we need to prove that the resulting object satisfies the appropriate logical relation. Ω is an element of the logical relation of term replacements. The attentive reader has probably already recognized that three cases might occur.

(1) A constructor has been encountered which is defined by Ω.
(2) A constructor has been encountered which has not been defined by Ω.
(3) A parameter has been encountered which must be defined in Ω.

The parameters in the third case are all local parameters of the iteration subject because initially, it is assumed to be closed. During the traversal, Ω is appropriately extended by new entries to map them to new parameters. Each of those three cases leads to a different lemma.

If we encounter the constructor $c : B$ defined in Σ, which is the domain of the logical relation for replacement, then $\langle\omega;\Omega\rangle(c)$ is in the logical relation $[\![\langle\omega\rangle(B)]\!]$. In the case that $c : B$ is not defined in this signature, then $\Omega(c)$ is undefined. In the case that the traversal of the iteration encounters a parameter $x : B$ defined in $\hat{\Psi}$, x is being renamed by the term replacement to a new variable name u, which is an element of $[\![\langle\omega\rangle(B)]\!]$ in context $\tilde{\Psi}$.

Lemma 7.12 (Access to logical relations for replacements)

(1) If $\Sigma = \Sigma_1, c : B, \Sigma_2$ and $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma;\hat{\Psi})]\!]$ then $\Psi \vdash M \in [\![\langle\omega\rangle(B)]\!]$ where $M = \langle\omega;\Omega\rangle(c)$
(2) If $\Sigma(c)$ is undefined and $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma;\hat{\Psi})]\!]$ then $\Omega(c)$ is undefined
(3) If $\hat{\Psi} = \hat{\Psi}_1, x : B, \hat{\Psi}_2$ and $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma;\hat{\Psi})]\!]$ then $\tilde{\Psi} \vdash u \in [\![\langle\omega\rangle(B)]\!]$ and $\tilde{\Psi} = \tilde{\Psi}_1, u : \langle\omega\rangle(B), \tilde{\Psi}_2$ where $u = \langle\omega;\Omega\rangle(x)$

Proof: by induction on the structure of $\hat{\Psi}$, Σ_2 in the first case, by induction on the structure of $\hat{\Psi}$, Σ in the second, and by induction on the structure of $\hat{\Psi}_2$ in the last (see [DPS97, Lemma 7.12-7.14]) □

The situation for matches is very similar. The argument follows the same pattern as for elimination. Recall from Definition 5.14 that the selection process traverses the case subject to find its head constructor. In the proof of Lemma 7.35 we need to show that the result of applying the match Ξ to the case subject satisfies the appropriate logical relation. Ξ is an element of the logical relation of matches. Contrary to the term replacement only two cases can occur, because the case object is well-typed and closed.

(1) A constructor is the head constructor which is defined in Ξ
(2) A parameter is the head constructor which is defined in Ξ.

First, if $c : B'$ is the head constructor, it is accounted for in Ξ and $\{B \Rightarrow A;\Xi;\Psi\}(c)$ is of correct type and an element in the logical relation $[\![\mathcal{C}\ (B, A, B')]\!]$.

Second, if $x : B'$ is the head constructor, there must be a match in Ξ, s.t. $\{B \Rightarrow A; \Xi; \Psi\}(x)$ is of correct type and an element in the logical relation $[\![\mathcal{C}\ (B, A, B')]\!]$.

Lemma 7.13 (Access to logical relations for matches)

(1) If $\Sigma = \Sigma_1, c : B', \Sigma_2$ and $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi})]\!]$ then $\Psi \vdash M \in [\![\mathcal{C}\ (B, A, B')]\!]$ where $M = \{B \Rightarrow A; \Xi; \Psi'\}(c)$ for an appropriate Ψ'.
(2) If $\hat{\Psi} = \hat{\Psi}_1, x : B', \hat{\Psi}_2$ and $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma; \hat{\Psi})]\!]$ then $\tilde{\Psi} \vdash u \in [\![\mathcal{C}\ (B, A, B')]\!]$ and $\tilde{\Psi} = \tilde{\Psi}_1, u : \mathcal{C}\ (B, A, B'), \tilde{\Psi}_2$ where $u = \{B \Rightarrow A; \Xi; \Psi'\}(x)$ for an appropriate Ψ'.

Proof: by induction on the structure of $\hat{\Psi}$, Σ_2 in the first case, by induction on the structure of $\hat{\Psi}_2$ in the second (see [DPS97, Lemma 7.15 and 7.16]) □

Two principal properties must be shown for the proof of the canonical form theorem via logical relations: First, every element of a logical relation has a canonical form, and second, every well-typed object is an element of the logical relation defined by its type, More formally:

$$\text{If } \Psi \vdash M \in [\![B]\!] \text{ then } \Psi \vdash M \Uparrow V : B \tag{1}$$
$$\text{If } \cdot; \Psi \vdash M : A \text{ then } \Psi \vdash M \in [\![A]\!] \tag{2}$$

All necessary lemmas are provided to show Property (1), the easier of those two properties. Before proving it, we must first generalize its formulation because the proof depends on the fact that atomic objects of pure type B are always in the logical relation of values $|B|$.

Lemma 7.14 (Logical relations and canonical forms)

(1) If $\Psi \vdash M \in [\![B]\!]$ then $\Psi \vdash M \Uparrow V : B$ for some V
(2) If $\Psi \vdash V \downarrow B$ then $\Psi \vdash V \in |B|$

Proof: by mutual induction on B, using Lemmas 6.1, 6.10, 6.12, 7.2, and 7.3 (see [DPS97, Lemma 7.17]) □

In the remainder of this section we show Property (2), which requires a long and complicated proof. To structure this proof, the remainder of this section is divided into several subsections, each describing new results which eventually form the proof of Theorem 7.36.

Each atomic object has necessarily pure type (as opposed to arbitrary types introduced in Section 3).

Lemma 7.15 (Types of atomic objects are pure)

If $\Psi \vdash V \downarrow A$ *then* A *is pure.*

Proof: by induction on the derivation of $\Psi \vdash V \downarrow A$ (see [DPS97, Lemma 7.18]) □

This result is needed to show that all objects in the relation of values $|A|$ are well-typed. The argument refers also to the definition of $[\![A]\!]$ just containing well-typed objects of type A.

Lemma 7.16 (Well-typedness of logical relations)

If $\Psi \vdash V \in |A|$ *then* $\cdot; \Psi \vdash V : A$

Proof: by induction on A, using Lemmas 6.11, 7.14, and 7.15 (see [DPS97, Lemma 7.19]) □

But this is not the only property objects satisfying relation $|A|$ enjoy. Based on the self evaluation Lemma 7.1, it is now easy to show that every object in $|A|$ evaluates to itself.

Lemma 7.17 (Logical relations: Self evaluation of values)

If $\Psi \vdash V \in |A|$ *then* $\Psi \vdash V \hookrightarrow V : A$

Proof: by induction on A, using Lemmas 7.1, 7.15, and 7.16 (see [DPS97, Lemma 7.20]) □

A direct consequence of the two previous lemmas is that every object in $|A|$ is also an object in $[\![A]\!]$. This is a result which we use quite frequently in the proofs of the subsequent lemmas.

Lemma 7.18 (Logical relation subsumption)

If $\Psi \vdash V \in |A|$ *then* $\Psi \vdash V \in [\![A]\!]$

Proof: direct, using Lemmas 7.16 and 7.17 (see [DPS97, Lemma 7.21]) □

It is necessary to show that every typable object of type A satisfies the relation $[\![A]\!]$. Consider a typing derivation ending with the typing rule TpApp. The result object of the rule is an application $M_1\ M_2$. M_1 is a function, M_2 is the argument of suitable type. The next lemma shows that it is legitimate to reason in a similar way with logical relations. If M_1 satisfies the logical relation $[\![A_2 \to A_1]\!]$ and M_2 satisfies $[\![A_2]\!]$, then $(M_1\ M_2)$ satisfies $[\![A_1]\!]$.

Lemma 7.19 (Logical relation is closed under application)

If $\Psi \vdash M_1 \in [\![A_2 \to A_1]\!]$ *and* $\Psi \vdash M_2 \in [\![A_2]\!]$ *then* $\Psi \vdash M_1\ M_2 \in [\![A_1]\!]$

Proof: direct, using Lemmas 7.2, 7.14, and 7.15 (see [DPS97, Lemma 7.22]) $\square$

The goal of this section is to show Property (2): If $\cdot; \Psi \vdash M : A$ then $\Psi \vdash M \in [\![A]\!]$. This property cannot be proven without generalization. The problem is that the context Ψ is not constant throughout a typing derivation (TpLam). The same observation holds for the modal context (TpLet).

We must hence generalize the formulation, using substitutions. Given a typing derivation $\Delta; \Gamma \vdash M : A$ and a substitution for $\Delta; \Gamma$, mapping variables from $\Delta; \Gamma$ to objects defined in a context Ψ, we can show that the substituted object $[\theta; \varrho](M)$ is indeed an object in $[\![A]\!]$. Property (2) is then a consequence of this result under the identity substitution.

7.4 Logical relations on substitutions

Let us continue with the description of the logical relation for contexts. The design of the logical relation is twofold, because the domain of a substitution $\theta; \varrho$ is a pair of contexts: one is the modal context Δ and the other is the arbitrary context Γ. This distinction is reflected in the design of the logical relation by the distinguishing two cases.

First, the logical relation for modal context contains all substitutions mapping variables of type A to closed objects in the logical relation $[\![A]\!]$.

Definition 7.20 (Logical relation for modal contexts) $\vdash \theta \in [\![\Delta]\!] :\Leftrightarrow$

Case: *If* $\Delta = \cdot$ *then* $\theta = \cdot$
Case: *If* $\Delta = \Delta', x : A$ *then* $\theta = \theta', M/x$ *and* $\cdot \vdash M \in [\![A]\!]$ *and* $\vdash \theta' \in [\![\Delta']\!]$

Second, the logical relation for arbitrary contexts contains all substitutions mapping variables of type A to objects in $|A|$. These objects need not to be closed. They may depend on new free variables from a context Ψ.

Definition 7.21 (Logical relation for regular contexts) $\Psi \vdash \varrho \in |\Gamma| :\Leftrightarrow$

Case: *If* $\Gamma = \cdot$ *then* $\varrho = \cdot$
Case: *If* $\Gamma = \Gamma', x : A$ *then* $\varrho = \varrho', V/x$ *and* $\Psi \vdash V \in |A|$ *and* $\Psi \vdash \varrho' \in |\Gamma'|$

As described earlier in this paper, we prefer to see the context Δ and the context Γ as one combined context. In this sense, it is useful to define a combined logical relation, which contains both, the logical relation for Δ and the one for Γ.

Definition 7.22 (Logical relation for combined contexts)

$$\Psi \vdash \theta; \varrho \in [\Delta; \Gamma] \;\; \textit{iff} \;\; \vdash \theta \in [\![\Delta]\!] \; \textit{and} \; \Psi \vdash \varrho \in |\Gamma|$$

The logical relations allow weakening on Ψ. We omit the easy proof by induction.

Lemma 7.23 (Weakening on logical relation on contexts)

(1) If $\Psi \vdash \varrho \in |\Gamma|$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash \varrho \in |\Gamma|$
(2) If $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ *and* $\Psi' \geq \Psi$ *then* $\Psi' \vdash \theta; \varrho \in [\Delta; \Gamma]$

Typing rule TpBox suggests that we need to restrict substitutions in $[\Delta; \Gamma]$ to substitutions in $[\Delta; \cdot]$, because boxed objects can only refer to variables in the modal context. Such a restriction is always possible: every substitution in $[\Delta; \Gamma]$ has the form $\theta; \varrho$, it can be restricted to $\theta; \cdot$.

Lemma 7.24 (Modal substitution restriction)

If $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ *then* $\cdot \vdash \theta; \cdot \in [\Delta; \cdot]$

Proof: direct (see [DPS97, Lemma 7.27]) □

Every substitution $\theta; \varrho$ which satisfies a logical relation $[\Delta; \Gamma]$ is well-formed with respect to Definition 6.4 (Typing of substitutions). In the proof we consider the modal part $\theta; \cdot$ and the regular part $\cdot; \varrho$ one by one. We show this property in four parts. First we show that $\theta; \cdot$ matches $\Delta; \cdot$, then we show that $\cdot; \varrho$ matches $\cdot; \Gamma$ in context Ψ, and third, that their combination $\theta; \varrho$ matches $\Delta; \Gamma$. It follows directly, that every substitution in a logical relation $[\Delta; \Gamma]$ matches the context pair $(\Delta; \Gamma)$.

Lemma 7.25 (Well-typedness of substitutions)

(1) If $\vdash \theta \in [\![\Delta]\!]$ *then* $\cdot; \cdot \vdash (\theta; \cdot) \; : \; (\Delta; \cdot)$
(2) If $\Psi \vdash \varrho \in |\Gamma|$ *then* $\cdot; \Psi \vdash (\cdot; \varrho) \; : \; (\cdot; \Gamma)$

(3) If $\cdot;\cdot \vdash (\theta;\cdot) \;:\; (\Delta;\cdot)$ *and* $\cdot;\Psi \vdash (\cdot;\varrho) \;:\; (\cdot;\Gamma)$ *then* $\cdot;\Psi \vdash (\theta;\varrho) \;:\; (\Delta;\Gamma)$
(4) If $\Psi \vdash \theta;\varrho \in [\Delta;\Gamma]$ *then* $\cdot;\Psi \vdash (\theta;\varrho) \;:\; (\Delta;\Gamma)$

Proof: by induction on Δ in the first case, by induction on Γ in the second case using Lemma 7.18, by induction on the derivation of $\cdot;\cdot \vdash (\theta;\cdot) \;:\; (\Delta;\cdot)$ in the third case, and the fourth case follows from the previous three. (see [DPS97, Lemma 7.28-7.31]) □

After this excursion into the basics of well-typed substitutions and their membership in logical relations, we focus now again on the proof of the canonical form theorem. With logical relations for contexts we can generalize Property (2).

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

To derive Property (2) from this generalized formulation it is necessary to show that $\Psi \vdash \cdot;\mathrm{id}_\Psi \in [\cdot;\Psi]$. Because of the definition of the logical relation for contexts $[\cdot;\Psi]$ two lemmas are necessary to prove it. First we show that for every Ψ, $\Psi \vdash \mathrm{id}_\Psi \in |\Psi|$ holds. This is not trivial, because the proof relies on Lemma 7.14. Then, we bring this lemma into the desired form: $\Psi \vdash \cdot;\mathrm{id}_\Psi \in [\cdot;\Psi]$.

Lemma 7.26 (Identity substitution for context) *For all* Ψ *the following holds:*

(1) $\Psi \vdash \mathrm{id}_\Psi \in |\Psi|$
(2) $\Psi \vdash \cdot;\mathrm{id}_\Psi \in [\cdot;\Psi]$

Proof: by induction on Ψ, using Lemmas 7.14 and 7.23 in the first case, from which the second case follows directly. (see [DPS97, Lemma 7.36-7.37]) □

Slowly we are approaching the canonical form theorem. Recall, that we are still trying to show that $\cdot;\Psi \vdash M : A$ implies that $\Psi \vdash M \in [\![A]\!]$. The logical relation for contexts has been the first step towards this lemma. Two more challenging problems must be tackled before we can finish its proof: the role of elimination and selection. They stem from the definition of the logical relation: an iterator or a case object, as any other object, can only be in the logical relation $[\![A]\!]$ if they evaluate to a value. For iteration (case) this implies that the result of elimination (selection) must evaluate to a value.

7.5 Auxiliary lemma for iteration

The elimination process is invoked during the evaluation of an iterator by the rule EvIt. Elimination traverses the structure of the canonical form of the iteration subject and replaces each parameter or constructor by its image under the term replacement Ω. Under the assumption that Ω is an element of the logical relation ($\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma'; \hat{\Psi})]\!]$) we can give the first preliminary formulation of the auxiliary lemma for iteration. $\hat{\Psi}$ represents the set of parameters possibly occurring in the canonical object, $\tilde{\Psi}$ contains their images under the term replacement Ω, and Ψ is the context in which the iterator object is well-typed.

If $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ then
$\hat{\Psi} \vdash V \Uparrow B'$ implies $\Psi, \tilde{\Psi} \vdash \langle\omega; \Omega\rangle(V) \in [\![\langle\omega\rangle(B')]\!]$

Canonical forms depend mutually on atomic forms, hence as second approximation we generalize this statement to:

If $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ then
(1) $\hat{\Psi} \vdash V \downarrow B'$ implies $\Psi, \tilde{\Psi} \vdash \langle\omega; \Omega\rangle(V) \in [\![\langle\omega\rangle(B')]\!]$, and
(2) $\hat{\Psi} \vdash V \Uparrow B'$ implies $\Psi, \tilde{\Psi} \vdash \langle\omega; \Omega\rangle(V) \in [\![\langle\omega\rangle(B')]\!]$.

But even this generalization does not go far enough. The rule CanLam introduces new parameters into the context $\hat{\Psi}$. To solve this predicament we introduce a substitution $\cdot; \mathrm{id}_\Psi, \varrho$ which is in the logical relation $[\![\cdot; \Psi, \tilde{\Psi}]\!]$. It is the identity on all variables from Ψ. This is a crucial property in the argument because it allows us to use the following *strengthening* lemma for the proof of the auxiliary lemma for iteration.

Lemma 7.27 (Strengthening lemma)

Let $\hat{\Delta}; \Gamma, \Gamma^*, \hat{\Gamma} \vdash (\mathrm{id}_{\hat{\Delta}}; \mathrm{id}_\Gamma, \varrho, \mathrm{id}_{\hat{\Gamma}}) \ : \ (\hat{\Delta}; \Gamma, \tilde{\Gamma}, \hat{\Gamma})$

(1) If $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash M : A$ *then* $M = [\mathrm{id}_{\hat{\Delta}}; \mathrm{id}_\Gamma, \varrho, \mathrm{id}_{\hat{\Gamma}}](M)$
(2) If $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash \Xi : \langle B \Rightarrow A\rangle(\Sigma')$ *then* $\Xi = [\mathrm{id}_{\hat{\Delta}}; \mathrm{id}_\Gamma, \varrho, \mathrm{id}_{\hat{\Gamma}}](\Xi)$
(3) If $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash \Omega : \langle\omega\rangle(\Sigma')$ *then* $\Omega = [\mathrm{id}_{\hat{\Delta}}; \mathrm{id}_\Gamma, \varrho, \mathrm{id}_{\hat{\Gamma}}](\Omega)$

Proof: by mutual induction on the derivations of $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash M : A$, $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash \Xi : \langle B \Rightarrow A\rangle(\Sigma')$, and $\hat{\Delta}; \Gamma, \hat{\Gamma} \vdash \Omega : \langle\omega\rangle(\Sigma')$, using Lemma 6.8 (see [DPS97, Lemma 7.38]) □

Consequently, the third generalization of the auxiliary lemma for iteration has the following form.

If $\Psi, \bar{\Psi} \vdash \cdot; \mathrm{id}_\Psi, \varrho \in [\cdot; \Psi, \tilde{\Psi}]$, $\Psi + \tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ then
(1) If $\hat{\Psi} \vdash V \downarrow B'$ then $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\langle\omega; \Omega\rangle(V)) \in [\![\langle\omega\rangle(B')]\!]$
(2) If $\hat{\Psi} \vdash V \Uparrow B'$ then $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\langle\omega; \Omega\rangle(V)) \in [\![\langle\omega\rangle(B')]\!]$

This formulation is very close to the version we will eventually prove. But we cannot prove it directly yet. The reason has been already discussed in Section 6. During the traversal, not every encountered constructor $c : C$ is an element of Σ': constructors whose target type $\tau(C)$ is not in $\mathcal{I}(\Sigma; B)$ are replaced by themselves according to the definition of $\mathsf{ElConst}$. Being not an element in $\mathcal{I}(\Sigma; B)$ can have two possible reasons when $\tau(C) \neq \tau(B)$:

(1) $\tau(C) \not\blacktriangleleft_{\Sigma;B} \tau(B)$
(2) $\tau(B) \not\blacktriangleleft_{\Sigma;B} \tau(C)$

In the first case we are done, as a consequence of Lemma 6.20 and Lemma 6.21 from Section 6. All what remains to show is that the second case cannot occur. It is possible to show that if during the elimination process a canonical form V of type B' is encountered then $\tau(B') \blacktriangleleft_{\Sigma;B} \tau(B)$ holds.

To prove this claim we define three conditions which are formulated as preconditions and postconditions for the elimination process — the auxiliary lemma for iteration will then be extended by these conditions. We distinguish between two preconditions: One for the case that the encountered object is canonical (*canonical precondition*) and the other for the case that the object is atomic (*atomic precondition*). If an atomic form is an application, then by inversion ElApp was the last applied rule. To show that the canonical precondition is satisfied for the right premiss, we must enforce a postcondition (*atomic postcondition*), which is valid after the the induction hypothesis was applied to the left premiss.

Definition 7.28 (Atomic precondition for elimination) *Let B an arbitrary pure type:*

$$Pre{\downarrow_B}\ (\Psi, B') :\Leftrightarrow \tau(B') \trianglelefteq_{\Sigma;B} \tau(B) \ \textit{and}\ \Psi \blacktriangleleft_{\Sigma;B} \tau(B)$$

The first part of this definition guarantees the weak subordination of $\tau(B')$ to $\tau(B)$. The second is necessary because the atomic precondition must imply the atomic postcondition, which finally might be used to show the canonical precondition. The canonical precondition is stronger than the atomic one. It states in addition that all source types of all parameter types of B' are subordinate to $\tau(B)$, as long as the parameters can be used in the definition of the object.

Definition 7.29 (Canonical precondition for elimination) *Let B arbitrary pure type:*

$$Pre{\Uparrow_B}\ (\Psi, B') :\Leftrightarrow$$

$$Pre{\downarrow_B}\ (\Psi, B') \ \textit{and for all}\ B'' \in \mathcal{P}(B') :$$

$$\textit{if } \tau(B'') \trianglelefteq_{\Sigma;B} \tau(B) \textit{ then for all } y \in \textit{Source}(B'') : y \blacktriangleleft_{\Sigma;B} \tau(B)$$

The atomic postcondition ensures that every source type of a type of the atomic object to be eliminated is actually subordinate to the target type of B — which might be very different from the target type of B'.

Definition 7.30 (Atomic postcondition for elimination) *Let B arbitrary pure type:*

$$\textit{Post}{\downarrow_B}\ (B') :\Leftrightarrow \textit{for all } y \in \textit{Source}(B') : y \blacktriangleleft_{\Sigma;B} \tau(B)$$

The proof of the auxiliary lemma for iteration is done by simultaneous induction over the atomic and canonical structure of the elimination subject. The following lemma shows that preconditions and postconditions imply each other in a suitable way as necessary to perform the inductive argument. For this purpose recall the definition of atomic and canonical forms from Definition 2.4. If we have a derivation ending in an application of AtVar and the atomic precondition holds then we must show that the postcondition holds. The same holds for a derivation ending with AtCon. In the case of application (AtApp) we encounter the following situation. Let $\mathcal{D}$ be a derivation ending in

$$\dfrac{\begin{array}{cc} \mathcal{D}_1 & \mathcal{D}_2 \\ \Psi \vdash V_1 \downarrow B_1 \rightarrow B_2 & \Psi \vdash V_2 \Uparrow B_1 \end{array}}{\Psi \vdash V_1\ V_2 \downarrow B_2}\ \textsf{AtApp}$$

By assumption we know that Pre $\downarrow_B$ (Ψ, B_2) holds. The application of the induction hypothesis to $\mathcal{D}_1$ requires that Pre $\downarrow_B$ $(\Psi, B_1 \rightarrow B_2)$ holds (to be proven). Finally for this case, the application of the induction hypothesis to $\mathcal{D}_2$ requires that Pre $\Uparrow_B$ (Ψ, B_1) holds. For this proof we must use Post $\downarrow_B$ $(B_1 \rightarrow B_2)$.

A complete list of all necessary implications is compiled in the following lemma. The first statement is needed for AtVar, the second for AtCon. The third and the forth are necessary for AtApp. CanAt does not require any special considerations, since the canonical precondition is stronger than the atomic one. The fifth statement makes the case CanLam go through. And finally the last fact provides the necessary information to ensure that the initial precondition holds (see Lemma 7.36).

Lemma 7.31 (Preservation of Preconditions and Postconditions)

(1) $\textit{Pre}{\downarrow_B}\ (\Psi, B')$ and $\Psi(x) = B'$ implies $\textit{Post}{\downarrow_B}\ (B')$

(2) $Pre{\downarrow_B}\ (\Psi, B')$ *and* $\Sigma(c) = B'$ *implies* $Post{\downarrow_B}\ (B')$
(3) $Pre{\downarrow_B}\ (\Psi, B_2)$ *implies* $Pre{\downarrow_B}\ (\Psi, B_1 \to B_2)$
(4) $Pre{\downarrow_B}\ (\Psi, B_2)$ *and* $Post{\downarrow_B}\ (B_1 \to B_2)$ *implies* $Pre{\Uparrow_B}\ (\Psi, B_1)$ *and* $Post{\downarrow_B}\ (B_2)$
(5) $Pre{\Uparrow_B}\ (\Psi, B_1 \to B_2)$ *implies* $Pre{\Uparrow_B}\ (\Psi, x : B_1, B_2)$
(6) *For all pure types B:* $Pre{\Uparrow_B}\ (\cdot, B)$

Proof: direct, using lemmas 6.13–6.18 (see [DPS97, Lemma 7.42]) □

Now all ingredients for the formulation of the auxiliary lemma for iteration are prepared. By inserting preconditions and postconditions into the previous formulation we obtain a provable lemma.

Lemma 7.32 (Auxiliary lemma for iteration)

If $\Psi, \bar{\Psi} \vdash \cdot; \mathrm{id}_\Psi, \varrho \in [\cdot; \Psi, \tilde{\Psi}]$, $\Psi{+}\tilde{\Psi} \vdash \Omega \in [\![\langle\omega\rangle(\Sigma'; \hat{\Psi})]\!]$ *and* $\Sigma' = \mathcal{S}^*(\Sigma; \mathcal{I}(\Sigma; B))$ *then*

(1) *If* $\hat{\Psi} \vdash V \downarrow B'$ *and* $Pre{\downarrow_B}\ (\hat{\Psi}, B')$ *then* $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\langle\omega; \Omega\rangle(V)) \in [\![\langle\omega\rangle(B')]\!]$ *and* $Post{\downarrow_B}\ (B')$
(2) *If* $\hat{\Psi} \vdash V \Uparrow B'$ *and* $Pre{\Uparrow_B}\ (\hat{\Psi}, B')$ *then* $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\langle\omega; \Omega\rangle(V)) \in [\![\langle\omega\rangle(B')]\!]$

Proof: by mutual induction on the derivations of $\hat{\Psi} \vdash V \Uparrow B'$ and $\hat{\Psi} \vdash V \downarrow B'$, using Lemmas 6.6, 6.19 – 6.20, 7.9 – 7.10, 7.12, 7.14, 7.18 – 7.19, 7.23, 7.25, 7.27, and 7.31 (see [DPS97, Lemma 7.43]) □

7.6 Auxiliary lemma for case

Similarly to the development of the auxiliary lemma for iteration we can prove an auxiliary lemma for case, which shows that the selection process always produces objects in the expected logical relation. The selection process is conceptually simpler than the elimination process, because only the head constructor is replaced by its image under the match. On the other hand a different kind of complexity is created by closing and boxing each argument.

For this purpose we must show that each canonical and each atomic object of type B is an element of the logical relation $[\![B]\!]$. CanLam may introduce new parameters in the derivation, hence we must generalize the first naive formulation and take substitutions $\cdot; \varrho$ into account.

Lemma 7.33 (Canonical elements and logical relations)

(1) *If* $\Psi \vdash V \downarrow B$ *and* $\Psi' \vdash \cdot; \varrho \in [\cdot; \Psi]$ *then* $\Psi' \vdash [\cdot; \varrho](V) \in [\![B]\!]$

(2) If $\Psi \vdash V \Uparrow B$ *and* $\Psi' \vdash \cdot; \varrho \in [\cdot; \Psi]$ *then* $\Psi' \vdash [\cdot; \varrho](V) \in [\![B]\!]$

Proof: by mutual induction on the derivations of $\Psi \vdash V \downarrow B$ and $\Psi \vdash V \Uparrow B$, using Lemmas 6.6, 7.14, 7.18 – 7.19, and 7.23 (see [DPS97, Lemma 7.44]) $\Box$

For the final preparatory lemma for the auxiliary lemma for case consider the rule SeApp from definition 5.14. The selection judgment applied to an application of the form $V_1\ V_2$ selects some object M_1 for V_1 and applies it to the boxed abstraction closure of V_2. V_2 can contain local parameters. We show that this abstraction closure is an object in the appropriate logical relation.

Lemma 7.34 (Abstraction closure)

If $\Psi \vdash V \Uparrow B$ *then* $\cdot \vdash \lambda\{\Psi\}. V \in [\![\Pi\{\Psi\}. B]\!]$

Proof: by induction on Ψ, using Lemmas 6.11 and 7.33 (see [DPS97, Lemma 7.45]) $\Box$

The development of the formulation of the auxiliary lemma for case is very similar to the iterator. Under the assumption that the match Ξ is an element of the logical relation ($\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \hat{\Psi})]\!]$) we need to prove (as a first approximation) that the resulting object of the selection process is in the logical relation $[\![\mathcal{C}^*\ (B, A, B)]\!]$. As in the formulation of the auxiliary lemma for iteration, $\hat{\Psi}$ is the set of parameters possibly occurring in the canonical object, $\tilde{\Psi}$ contains their images under the match Ξ, and Ψ is the context in which the case object is well-typed.

If $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}(\Sigma; \tau(B))$ then
  If $\hat{\Psi} \vdash V \Uparrow B'$ then $\Psi, \tilde{\Psi} \vdash \{B \Rightarrow A; \Xi; \hat{\Psi}\}(V) \in [\![\mathcal{C}^*\ (B, A, B')]\!]$

As in the lemma for iteration this formulation of the lemma cannot be proven without further generalization. Canonical forms depend mutually on atomic forms and with a careful distinction of "inner case types" and "outer case types" we can refine this statement in the following way.

If $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}(\Sigma; \tau(B))$ then
(1) If $\hat{\Psi} \vdash V \downarrow B'$ then $\Psi, \tilde{\Psi} \vdash \{B \Rightarrow A; \Xi; \hat{\Psi}\}(V) \in [\![\mathcal{C}\ (B, A, B')]\!]$
(2) If $\hat{\Psi} \vdash V \Uparrow B'$ then $\Psi, \tilde{\Psi} \vdash \{B \Rightarrow A; \Xi; \hat{\Psi}\}(V) \in [\![\mathcal{C}^*\ (B, A, B')]\!]$

The rule CanLam poses the same problems as in the iteration case. The substitution $\cdot; \mathrm{id}_\Psi, \varrho$ leads to a further generalization of the auxiliary lemma for case.

If $\Psi, \bar{\Psi} \vdash \cdot; \mathrm{id}_\Psi, \varrho \in [\cdot; \Psi, \tilde{\Psi}]$, $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \hat{\Psi})]\!]$ and $\Sigma' = \mathcal{S}(\Sigma; \tau(B))$ then
(1) If $\hat{\Psi} \vdash V \downarrow B'$ then $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\{B \Rightarrow A; \Xi; \hat{\Psi}\}(V)) \in [\![\mathcal{C}\ (B, A, B')]\!]$

(2) If $\hat{\Psi} \vdash V \Uparrow B'$ then $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\{B \Rightarrow A; \Xi; \hat{\Psi}\}(V)) \in [\![\mathcal{C}^* \ (B, A, B')]\!]$

This formulation is very close to the version of the lemma we will prove. But we cannot prove it directly, yet. During the proof, we must recover the initial type B. This can only be done by assuming for the atomic case that $\Pi\{\hat{\Psi}\}.\, \tau(B) = B$ and also $\tau(B) = \tau(B')$. For the canonical case, we can assume that $\hat{\Psi}$ represents the initial set of domain types of type B. The remaining domain types are still contained as abstractions in the type B': $\Pi\{\hat{\Psi}\}.\, B' = B$.

Lemma 7.35 (Auxiliary lemma for case)

If $\Psi, \bar{\Psi} \vdash \cdot; \mathrm{id}_\Psi, \varrho \in [\cdot; \Psi, \tilde{\Psi}]$, $\Psi + \tilde{\Psi} \vdash \Xi \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \hat{\Psi})]\!]$ *and* $\Sigma' = \mathcal{S}(\Sigma; \tau(B))$ *then*

(1) If $\hat{\Psi} \vdash V \downarrow B'$ *and* $\Pi\{\hat{\Psi}\}.\, \tau(B) = B$ *and* $\tau(B') = \tau(B)$ *then* $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\{B \Rightarrow A; \Xi; \hat{\Psi}\}(V)) \in [\![\mathcal{C}\ (B, A, B')]\!]$
(2) If $\hat{\Psi} \vdash V \Uparrow B'$ *and* $\Pi\{\hat{\Psi}\}.\, B' = B$ *then* $\Psi, \bar{\Psi} \vdash [\cdot; \mathrm{id}_\Psi, \varrho](\{B \Rightarrow A; \Xi; \hat{\Psi}\}(V)) \in [\![\mathcal{C}^* \ (B, A, B')]\!]$

Proof: by mutual induction on the derivations of $\Psi \vdash V \downarrow B$ and $\Psi \vdash V \Uparrow B$, using Lemmas 6.6, 7.9, 7.11, 7.13 – 7.14, 7.18 – 7.19, 7.23, 7.25, 7.27, and 7.34 (see [DPS97, Lemma 7.46]) □

7.7 Canonical form theorem

This concludes the presentation of the preliminary properties. All the ingredients are prepared and await to be put together to prove Property (2). Recall that the proof of the canonical form theorem is performed in two steps: The first step we generalized already once (which led to the introduction of logical relations for contexts), and the second we already completed (in Lemma 7.14).

(1) If $\Delta; \Gamma \vdash M : A$ and $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$ then $\Psi \vdash [\theta; \varrho](M) \in [\![A]\!]$
(2) If $\Psi \vdash M \in [\![B]\!]$ then $\Psi \vdash M \Uparrow V : B$

To complete the proof of the first property we must generalize it to term replacements and matches, leading to the following lemma. It is the centerpiece of this work because its proof combines all results obtained so far.

Lemma 7.36 (Typing and logical relations)

Let $\Psi \vdash \theta; \varrho \in [\Delta; \Gamma]$

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

(2) If $\Delta; \Gamma \vdash \Omega : \langle\omega\rangle(\Sigma')$ *then* $\Psi + \cdot \vdash [\theta; \varrho](\Omega) \in [\![\langle\omega\rangle(\Sigma'; \cdot)]\!]$
(3) If $\Delta; \Gamma \vdash \Xi : \langle B \Rightarrow A\rangle(\Sigma')$ *then* $\Psi + \cdot \vdash [\theta; \varrho](\Xi) \in [\![\langle B \Rightarrow A\rangle(\Sigma'; \cdot)]\!]$

Proof: by mutual induction on the derivation of $\Delta; \Gamma \vdash M : A$, using Lemmas 6.6, 6.9, 7.2, 7.6, 7.8 – 7.9, 7.14, 7.18 – 7.19, 7.24, 7.25, 7.32, and 7.35 (see [DPS97, Lemma 7.47]) □

The canonical form theorem is only a simple consequence of Lemma 7.14 and Lemma 7.36. It says, that every object M of pure type evaluates to a canonical form. In other words, no matter how complex the form of the object M is, it may contain λ-abstractions, applications, boxes, and lets, it will always evaluate to a canonical form, only containing λ-abstractions and applications. Section 9 emphasizes this point again and shows the usefulness of this result.

Theorem 7.37 (Canonical form theorem)

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

Proof: direct, using Lemmas 7.14, 7.26, and 7.36 (see [DPS97, Lemma 7.48])

□

8 Type preservation theorem

The canonical form theorem is a very powerful theorem. The type preservation property for the operational semantics of our system follows as a corollary if evaluations are deterministic. This claim is intuitively immediate because the form of the object triggers the evaluation rule — which is uniquely defined. The operational semantics depends mutually on the evaluation to canonical forms, hence the uniqueness lemma reads as follows.

Lemma 8.1 (Uniqueness of evaluation)

(1) If $\Psi \vdash M \Uparrow V : A$ *and* $\Psi \vdash M \Uparrow V' : A$ *then* $V = V'$
(2) If $\Psi \vdash M \hookrightarrow V : A$ *and* $\Psi \vdash M \hookrightarrow V' : A$ *then* $V = V'$

Proof: by mutual induction on the derivations of $\Psi \vdash M \Uparrow V : A$ and $\mathcal{E} :: \Psi \vdash M \hookrightarrow V : A$ (see [DPS97, Lemma 8.1]) □

Together with Lemma 7.36 the type preservation theorem follows directly.

Theorem 8.2 (Type preservation)

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

Proof: direct, using Lemmas 7.18, 7.26, 7.36, and 8.1 (see [DPS97, Lemma 8.2]) □

In the next section we present another corollary from Lemma 7.36: The modal λ-calculus is a conservative extension of the simply-typed λ-calculus.

9 Conservative extension theorem

By the definition of the modal λ-calculus, it is clear that the language of objects and types extends the language of the simply-typed λ-calculus. It follows quite naturally that every typing derivation in the simply-typed calculus can be represented in our system: Using the empty modal context, StpVar must be replaced by TpVarR, StpConst by TpCon, StpLam by TpLam, and finally StpApp by TpApp.

Lemma 9.1 (Typing extension)

If $\Psi \vdash M : B$ *then* $\cdot; \Psi \vdash M : B$

Proof: by induction over $\Psi \vdash M : B$ (see [DPS97, Lemma 9.1]) □

Let M be an object of pure type B with free variables from a pure context Ψ. M itself need not to be pure but rather some object in the modal λ-calculus using boxes, lets, iterators, and definition by cases. We have seen that M has a canonical form V, and Lemma 7.2 (1) shows that V must be an object in the simply-typed λ-calculus.

Theorem 9.2 (Conservative Extension)

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

Proof: direct, using Lemma 7.2, and Theorem 7.37 (see [DPS97, Lemma 9.2]) □

This concludes the discussion of the meta-theoretic properties of the modal λ-calculus.

10 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 [Mil92,Pfe91].

Our work was inspired by Miller's system [Mil90], 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 [MH95] and its refinement by Fegaras and Sheard [FS96] 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 [FS96] 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 [CNSvS94] or proposed by Jouannaud and Okada [JO91] 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 [2] 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. We also plan to reexamine applications in the realm of functional programming [Mil90,FS96] and related work on reasoning about higher-order abstract syntax with explicit induction [DH94,DFH95] or definitional reflection [MM97].

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

[Chu40] Alonzo Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[CNSvS94] 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.

[DFH95] 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.

[DH94] 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.

[DP96] 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.

[DPS97] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In R. Hindley, editor, *Third*

[2] personal communication

International Conference on Typed Lambda Calculi and Applications, Nancy, France. Springer Verlag, April 1997. See also Technical Report CMU-CS-96-172, Carnegie Mellon University, September 1996.

[FS96] 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.

[Göd90] 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.

[HHP93] 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.

[JO91] 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.

[Mag95] 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.

[MH95] 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.

[Mil90] 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.

[Mil91] 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.

[Mil92] 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.

[MM97] Raymond McDowell and Dale Miller. A logic for reasoning about logic specifications. In Glynn Winskel, editor, *Proceedings of the Twelfth Annual Symposium on Logic in Computer Science*. IEEE Computer Society Press, June 1997. 1997.

[NPS90] 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.

[Ode94] Martin Odersky. A functional theory of local names. In *Proceedings of 21st Annual Symposium on Principles of Programming Languages*, pages 48–59, Portland, Oregon, January 1994. ACM Press.

[Pfe91] 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.

[PM93] 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.

[PW95] 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 Semantics*, New Orleans, Louisiana, March 1995. To appear in *Electronic Notes in Theoretical Computer Science*, Volume 1, Elsevier.