Programming with Higher-Order Abstract Syntax [*]

Carsten Schürmann
Department of Computer Science
Yale University
New Haven, CT 06520

ABSTRACT

We describe the $\lambda^{\Rightarrow\star\Box}$-calculus that permits function definition by cases over objects of arbitrary simple (functional) type. In $\lambda^{\Rightarrow\star\Box}$ higher-order abstract syntax and elimination forms for inductive types safely coexist. $\lambda^{\Rightarrow\star\Box}$ is a two tier system with a modal type that connects the logical framework to the programming language. We illustrate the expressive power and elegance of $\lambda^{\Rightarrow\star\Box}$ using the example of parallel reduction.

1. INTRODUCTION

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

This deceptively simple idea, which goes back to Church [1] and Martin-Löf's system of arities [12], 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 elimination techniques do not apply.

In recent years, programming language designers have proposed many extensions to the λ-calculus to preserve the advantages of higher-order abstract syntax while allowing elimination forms for inductively defined datatypes. Despreyroux's proposal [4, 3] distinguishes between types for parameters and expressions, but could not solve the problem of exotic terms. The modal λ-calculus with iteration by Schürmann, Despeyroux, and Pfenning [15] is a theory of primitive recursive functionals that decomposes the primitive recursive function space $A \Rightarrow B$ into a modal operator and a parametric function space $(\Box A) \rightarrow B$. The modal λ-calculus, although theoretically elegant, is lacking an implementation partly because the proposed form of iteration is very rigid and somewhat clumsy.

Gabbay and Pitts use FM-sets [14, 8] to introduce a notion of α-equivalence to functional programming, however, other forms of definitional equality including β and η are left to the programmer (should he or she need them).

In this paper, we make another attempt to reconcile higher-order abstract syntax under $\alpha\beta\eta$ with function definition by cases. The central contribution is the $\lambda^{\Rightarrow\star\Box}$-calculus, which embeds the simply-typed $\lambda^{\rightarrow\times}$-calculus via a modal $\Box$ operator. A *world* system enforces the consistency of the binding occurrences and the uses of parameters. The modality separates the parametric function space $\Box(A \rightarrow B)$ from the function space $(\Box A) \Rightarrow (\Box B)$ for programming.

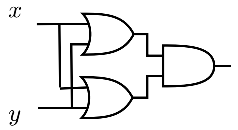
$\lambda^{\rightarrow\times}$ is a logical framework that serves the (higher-order) representation of data. The only computational rule on terms is β-reduction. $\lambda^{\Rightarrow\star\Box}$ on the other hand, is also simply typed, but provides in addition operators for matching and recursion. In $\lambda^{\Rightarrow\star\Box}$ it is possible, for example, to recurse over the structure of expressions from the untyped λ-calculus when represented using higher-order abstract syntax, and compute under λ-binders. Case analysis can be performed over *any* $\lambda^{\rightarrow\times}$-object, even those of functional type. $\lambda^{\Rightarrow\star\Box}$ satisfies the usual meta-theoretic requirement of a λ-calculus for programming. It is type preserving. By design, $\lambda^{\Rightarrow\star\Box}$ is not strongly normalizing, since it contains an explicit recursion operator.

The remainder of this paper is organized as follows: In Section 2, we advertise higher-order abstract syntax, using Boolean circuits as example. Section 3 reviews the idea of higher-order abstract syntax and introduces the simply typed $\lambda^{\rightarrow\times}$-calculus. In Section 4 and Section 5 we describe the type theory $\lambda^{\Rightarrow\star\Box}$ and its operational semantics, respectively. For explanatory reasons one of the rules is unnecessarily restricted. A generalized version is then discussed in Section 6. Section 7 gives some more examples and sheds some light on the expressive power of $\lambda^{\Rightarrow\star\Box}$, before we discuss its meta-theory in Section 8. Finally, we mention some more related work in Section 9 and conclude in Section 10.

2. MOTIVATION

[*]This research is funded by NFS under grant CCR-0133502.

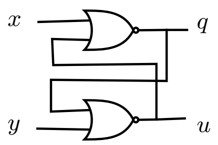
Sequential Boolean circuits are often taught in introductory classes to explain the theoretical foundations of computer science. A circuit consists of many interconnected gates, including $\wedge$, $\neg$, and $\vee$ gates. A Boolean circuit is called *sequential*, if it contains feed-back loops, as for example, a flip-flop element, or a differential multiplier. For functional programmers, the first idea to come to mind is probably to take represent circuits as datatypes. One constructor to consider is `CONNECTOR2` that maps two input lines to one output line. `AND`, `OR` stand for gates.



An ML declaration that corresponds to this circuit is

```
CONNECTOR2 (fn x => fn y =>
  AND (OR (x, y), OR (x, y))).
```

If we also add sequentiality, in form of a `LOOP`, and a `CONNECT2` to connect two circuits to a circuit that has two input lines, a flip-flop element



is encoded in ML as

```
LOOP (fn c =>
  CONNECTOR2 (fn q => fn u =>
    CONNECTOR2 (fn x => fn y =>
      CONNECT2 (NOT (AND (x, u)),
                NOT (AND (y, q))), c)))).
```

The following datatype declaration captures the essence of circuits that take m input lines and return at most one output line. A similar datatype could be given for Haskell or any other language providing datatypes.

```
datatype Circuit
  = CONNECT1   of Circuit * Circuit
  | CONNECTOR1 of (Circuit -> Circuit)
  | CONNECT2   of Circuit * Circuit * Circuit
  | CONNECTOR2 of (Circuit -> Circuit -> Circuit)
  | NOT        of Circuit
  | AND        of Circuit * Circuit
  | NAND       of Circuit * Circuit
  | LOOP       of (Circuit -> Circuit)
  | TRUE
  | FALSE
```

This datatype employs higher-order abstract syntax, and is hence not inductive. But it is convenient to use. For example, it is not necessary to create and maintain names for input or output lines and connectors, to maintain the topology of the circuit explicity, and to model the transport of information across lines. All this functionality is automatically provided by the variable mechanism built into the host language, such as OCaml, SML, or Haskell. And therefore, some functions that work on circuits are particularly easy to write, for example measuring the size of a circuit, the depth, or even simulating a circuit.

However, the use of higher-order abstract syntax in a functional language has its limitations. First, the encoding might not be adequate, which means that there are elements of type Circuit that do not correspond to real circuits, such as

```
CONNECTOR1 (fn _ => raise Domain).
```

When adequacy is broken, the responsibility of maintaining the soundness of representations is returned to the programmer. Second, there are many functions one would like to write but cannot. Examples include testing two circuits for equivalence, converting one circuit into an equivalent one that only uses $\overline{\wedge}$-gates, or minimizing the number of gates inside a circuit, etc. Functions of this kind cannot be evaluated since evaluation under function binders is usually prohibited in functional languages.

The function `convert : Circuit -> Circuit` is an ill-fated attempt to convert a circuit into another circuit that only uses $\overline{\wedge}$-gates in SML.

```
fun convert (CONNECT1 (c1, c2)) =
      CONNECT1 (convert c1, convert c2)
  | convert (CONNECTOR1 c) = <problem>
  | convert (CONNECT2 (c1, c2, c3)) =
      CONNECT2 (convert c1, convert c2, convert c3)
  | convert (CONNECTOR2 c) = <problem>
  | convert (NOT c) =
      CONNECT1 (map convert c,
      CONNECTOR1 (fn x => NAND (x, x)))
  | convert (AND (c1, c2)) =
      CONNECT1 (convert c1,
      CONNECTOR1 (fn x1 =>
      CONNECT1 (convert c2,
      CONNECTOR1 (fn x2 =>
      CONNECT1 (NAND (x1, x2),
      CONNECTOR1 (fn x3 => NAND (x3, x3)))))))
  | convert (LOOP c) = <problem>
  | convert (TRUE) = TRUE
  | convert (FALSE) = FALSE
```

Unfortunately, there is no obvious way to spell out the cases marked by `<problem>`. In Section 7, we will return to this particular problem and show its solution in Example 12.

3. HIGHER-ORDER ABSTRACT SYNTAX

Higher-order abstract syntax exploits the full expressive power of the simply-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 the simply typed λ-calculus, although we recognize, that an extension to dependent types and polymorphism is important future work. Our formulation of the simply-typed $\lambda^{\rightarrow\times}$-calculus with pairs is standard.

$$
\begin{array}{llcl}
\text{Types:} & A & ::= & a \mid A_1 \rightarrow A_2 \mid A_1 \times A_2 \\
\text{Objects:} & M & ::= & x \mid M_1 \cdot M_2 \mid \lambda x : A.\, M \\
 & & & \mid \langle M_1; M_2 \rangle \mid \pi_1 M \mid \pi_2 M \\
\text{Contexts:} & \Gamma & ::= & \cdot \mid \Gamma, x : A \\
\text{Type signature:} & \Sigma & ::= & \cdot \mid \Sigma, a
\end{array}
$$

We write a for type level constants, and x for variables. We do not use object level constants. Instead, we assume all available constants to be declared in the context. In the absence of dependencies, two contexts are considered equivalent modulo exchange. Instead of juxtaposition, we write $M_1 \cdot M_2$ for application. A type signature Σ contains the declarations of type level constants. Since Σ will always be clear from the context, we omit it from all typing and evaluation rules.

Without constants, the present λ-calculus can hardly be considered a logical framework, or meta-language for representing syntax. Logical frameworks typically allow the signature to declare type level constants a, and object level constants c, that *are* the elementary building blocks or constructors to objects.

For reasons that become clear later, we do not wish to make a difference between parameters and object level constants. Constants can be viewed as parameters, that are buried deeply inside a context. What kind of constants can be found inside a context is summarized by a concept that we call *world.* A world can be seen as the "type" of a context, local parameters excluded.

$$w \quad ::= \quad \cdot \mid A \mid w_1 + w_2$$

We write $\cdot$ for the empty world, in which no constants are permitted at all, only local parameters. Every type A is a world. $w_1 + w_2$ describes all context providing constants for w_1 or w_2.

Example 1 (Circuits). *Let*

$$\begin{array}{rcl}
\mathsf{Circuit} & = & (\mathsf{circuit} \times \mathsf{circuit} \to \mathsf{circuit}) \\
& & \times\, ((\mathsf{circuit} \to \mathsf{circuit}) \to \mathsf{circuit}) \\
& & \times\, ((\mathsf{circuit} \to \mathsf{circuit} \to \mathsf{circuit}) \to \mathsf{circuit}) \\
& & \times\, (\mathsf{circuit} \to \mathsf{circuit}) \\
& & \times\, (\mathsf{circuit} \times \mathsf{circuit} \to \mathsf{circuit}) \\
& & \times\, (\mathsf{circuit} \times \mathsf{circuit} \to \mathsf{circuit}) \\
& & \times\, ((\mathsf{circuit} \times \mathsf{circuit}) \to \mathsf{circuit}) \\
& & \times\, \mathsf{circuit} \\
& & \times\, \mathsf{circuit}.
\end{array}$$

$\mathsf{Circuit}$ *is a defined type and hence a world. The projections of a term of type* $\mathsf{Circuit}$ *are the constructors for the type* $\mathsf{circuit}$. *In the remainder of the paper, we will use lowercase names to stand for declared types and capitalized for defined types.*

There is a close relation between type level constants, worlds, and the usual meaning of a datatype. For datatypes that do not use polymorphism or rely on non-parametric functional arguments, standard datatypes are indeed subsumed by our definition. But the world system is more general than that. Multiple declarations of the same world A are permitted in one context. Repetition of this kind is important, because we will use it to capture stacks of function binders under which evaluation takes place. In the the circuit example, the world $\mathsf{Circuit} + \mathsf{circuit}$ permits contexts that contain the constructors for circuits as well as multiple parameters of type $\mathsf{circuit}$.

under several functional arguments of type $\mathsf{circuit}$.

Example 2 (Constructors). *For any declaration of type* $o : \mathsf{Circuit}$ *we use the following abbreviations to refer to the constructors contained in* o*:*

$$\begin{array}{lcl}
\mathsf{connect} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ o \\
\mathsf{connector1} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2\ o) \\
\mathsf{connector2} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^2\ o) \\
\mathsf{not} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^3\ o) \\
\mathsf{and} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^4\ o) \\
\mathsf{nand} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^4\ o) \\
\mathsf{loop} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^5\ o) \\
\mathsf{true} & = & \lambda o : \mathsf{Circuit}.\, \pi_1\ (\pi_2^6\ o) \\
\mathsf{false} & = & \lambda o : \mathsf{Circuit}.\, \pi_2^7\ o
\end{array}$$

We write o *for variables that range over a collection of constructors that are somewhat reminiscent of records. In this vain we use the syntactic sugar* $o.c$ *to stand for* $c \cdot o$. *The flip-flop circuit in* $\lambda^{\to\times}$ *has the following form.*

$$\begin{array}{rl}
\mathrm{flipflop}\ o = & o.\mathsf{loop}(\lambda r : \mathsf{circuit}. \\
& o.\mathsf{connector}(\lambda q : \mathsf{circuit} \times \mathsf{circuit}. \\
& o.\mathsf{connector}(\lambda x : \mathsf{circuit} \times \mathsf{circuit}. \\
& o.\mathsf{connect}(\langle o.\mathsf{not} \cdot (o.\mathsf{and} \cdot (\pi_1\ x) \cdot (\pi_2\ q)); \\
& \qquad\qquad o.\mathsf{not} \cdot (o.\mathsf{and} \cdot (\pi_2\ x) \cdot (\pi_1\ q))\rangle\ r))))
\end{array}$$

and we can easily see that $o : \mathsf{Circuit} \vdash_w \mathrm{flipflop}\ o : \mathsf{circuit}$.

Definition 3 (Typing). *We write* $\Gamma \vdash_w M : A$ *for the typing judgment.* w *is the world, and* Γ *may contain none, one, or several declarations of constructors for each type in* w. *The typing rules are given in Figure 1. (The index* w *to the judgment is optional.)*

We write $[N/x]M$ for the substitution of term N in M for x. As usual, we assume that substitutions are capture avoiding by tacitly renaming variables.

In this paper, we fix the underlying definitional equational theory to $\beta\eta$ [2]. However, it is true that the results presented here only depend on the existence of canonical forms, i.e. every term has a unique normal form, and therefore different forms of definitional equality are imaginable. Canonical forms play the role of observable values that are in a one-to-one correspondence with the object language we are trying to represent.

$$(\lambda x : A.\, M_1) \cdot M_2 \quad \equiv \quad M_1[M_2/x] \qquad (1)$$

$$\lambda x : A.\, M \cdot x \quad \equiv \quad M \qquad (2)$$

We base our choice of canonical form on β-normal, η-long forms because of the following well-known theorem.

Theorem 4 (Canonical form). *Let* $\Gamma \vdash_w M : A$ *be a term in* $\lambda^{\to\times}$. *There exists a canonical form* $\Gamma \vdash_w M' : A$, *such that* $M \equiv M'$.

We summarize a few well-known properties of the simply typed λ-calculus.

Theorem 5 (Isomorphism). *Let* $\mathcal{D}$ *be a derivation of* $\Gamma \vdash_w M : A$. *For a fixed* Γ, *there exists an isomorphism between* $\mathcal{D}$ *and* M.

Proof. By structural induction on the derivation $\mathcal{D}$. $\square$

Theorem 6 (Weakening). *Let* $\mathcal{D}$ *be a derivation of* $\Gamma \vdash_w M : A$ *then* $\Gamma, x : B \vdash_w M : A$.

Proof. By structural induction on $\mathcal{D}$. $\square$

Lemma 7 (Substitution). *Let* $\mathcal{D}$ *be a derivation of* $\Gamma, x : A \vdash_w M : B$ *and* $\Gamma \vdash_w N : A$ *then* $\Gamma \vdash_w [N/x]M : B$

Proof. By structural induction on $\mathcal{D}$. $\square$

$$\frac{\Gamma(x) = A}{\Gamma \vdash_w x : A}\ \mathsf{var} \qquad \frac{\Gamma, x : A_1 \vdash_w M : A_2}{\Gamma \vdash_w \lambda x : A_1.\, M : A_1 \to A_2}\ \mathsf{lam} \qquad \frac{\Gamma \vdash_w M_1 : A_2 \to A_1 \quad \Gamma \vdash_w M_2 : A_2}{\Gamma \vdash_w M_1 \cdot M_2 : A_1}\ \mathsf{app}$$

$$\frac{\Gamma \vdash_w M_1 : A_1 \quad \Gamma \vdash_w M_2 : A_2}{\Gamma \vdash_w \langle M_1; M_2 \rangle : A_1 \times A_2}\ \mathsf{pair} \qquad \frac{\Gamma \vdash_w M : A_1 \times A_2}{\Gamma \vdash_w \pi_1\, M : A_1}\ \mathsf{fst} \qquad \frac{\Gamma \vdash_w M : A_1 \times A_2}{\Gamma \vdash_w \pi_2\, M : A_2}\ \mathsf{snd}$$

Figure 1: Type system for the $\lambda^{\to\times}$.

4. TYPE THEORY $\lambda^{\Rightarrow\star\Box}$

Circuits are not inductively definable, because the types of loop, connector1, connector2, contain negative occurrences of type circuit. This is not just a formal observation, but has practical consequences. We cannot formulate a consistent elimination form for circuits in this representation. If we increase the expressive power of $\lambda^{\to\times}$ by a case construct or an iterator, not every well-typed object has a canonical form, giving raise to so called exotic terms and ultimately breaking adequacy.

$$\mathsf{connect1}\ (\lambda c : \mathsf{circuit}.\, \mathrm{case}\ x\ \ \mathrm{of}\ \mathsf{true} \Rightarrow \mathsf{false} | \_ \Rightarrow \mathsf{true})$$

The $\lambda^{\Rightarrow\star\Box}$-calculus that we discuss next solves the problem of in an elegant way. We should think of $\lambda^{\Rightarrow\star\Box}$ as another simply-typed λ-calculus that supports function definition by cases and recursion and embeds $\lambda^{\to\times}$ using the $\Box$ modality. A parametric function has type $\Box(A \to B)$ and a function for programming $\Box A \Rightarrow \Box B$.

Intuitively, $\Box A$ stands for a typing derivation $\mathcal{D} :: \Gamma \vdash M : A$ as a whole, and not just an object M of type A. This is the critical idea underlying this work. The entire type system of $\lambda^{\Rightarrow\star\Box}$ can be directly derived from this intuition.

As a motivating example, consider writing a function f_c for circuits (see Example 1) that replaces all occurrences of $\wedge$- and $\neg$-gates in a circuit by $\overline{\wedge}$-gates. Its type reflects that a derivation $\mathcal{D} :: \Gamma \vdash_w M : \mathsf{circuit}$ is mapped into a typing derivation $\mathcal{D}' :: \Gamma \vdash_w M' : \mathsf{circuit}$. The first attempt to assign a type to f_c is

$$\Box(\Gamma \vdash_w M : \mathsf{circuit}) \Rightarrow \Box(\Gamma \vdash_w M' : \mathsf{circuit}).$$

By Theorem 5 there exists an isomorphism between typing derivations and terms. M and M' can be respectively reconstructed in $\lambda^{\to\times}$ from $\mathcal{D}$ and $\mathcal{D}'$, because every typing rule is endowed with a unique term constructor. We may thus drop M from the function type

$$f_c \in \Box(\Gamma \vdash_w \mathsf{circuit}) \Rightarrow \Box(\Gamma \vdash_w \mathsf{circuit}).$$

Without loss of generality, all contexts appearing in a type can be assumed identical. If not, using Theorem 6 the contexts that are too small may be weakened. Since weakening is admissible, and the world w of Γ is known, Γ itself is actually unimportant, and can hence also be omitted.

$$f_c \in \langle w; \Box\mathsf{circuit} \Rightarrow \Box\mathsf{circuit} \rangle.$$

The $\langle\cdot;\cdot\rangle$ type constructor "closes" a type with respect to the world in which it is valid. In summary, we distinguish between types and closed types.

$$\frac{}{\vdash A \in A}\ \mathsf{id} \qquad \frac{\vdash A \in w_1}{\vdash A \in w_1 + w_2}\ \mathsf{left} \qquad \frac{\vdash A \in w_2}{\vdash A \in w_1 + w_2}\ \mathsf{right}$$

$$\frac{}{\vdash \cdot \in w}\ \mathsf{empty} \qquad \frac{\vdash \Gamma \in w \quad \Gamma \vdash A \in w}{\vdash \Gamma, A \in w}\ \mathsf{cons}$$

Figure 2: Valid (well-worlded) context.

Types	$\tau ::= \Box A \mid \tau_1 \Rightarrow \tau_2 \mid \tau_1 \star \tau_2$
Closed Types	$\sigma ::= \langle w; \tau \rangle$
Contexts	$\Phi ::= \cdot \mid \Phi, x : \tau$
Closed Contexts	$\Psi ::= \cdot \mid \Psi, u \in \sigma$

We write x for variables ranging over expressions, and u for variables ranging over closed expressions. $\tau_1 \star \tau_2$ is the usual pair type that we need to express mutually recursive functions in $\lambda^{\Rightarrow\star\Box}$ (see Example 14).

The $\lambda^{\Rightarrow\star\Box}$-calculus provides a modal operator $\Box$, which is exclusively used to connect it to the $\lambda^{\to\times}$-calculus. The modality is necessarily degenerate, since the type $A \Rightarrow \Box A$ is syntactically invalid. The distribution axiom of modal logic, however, is still satisfied (see Lemma 15). The sole purpose of $\Box A$ is to embed $\lambda^{\to\times}$ into $\lambda^{\Rightarrow\star\Box}$.

The typing judgment for expressions of $\lambda^{\Rightarrow\star\Box}$ has four zones.

$$\Gamma; \Delta; \Phi; \Psi$$

The two left most zones Γ and Δ belong to $\lambda^{\to\times}$. Γ collects all universal variables, and Δ the existential ones. Variables declared in Γ cannot be analyzed by cases, but assumptions in Δ may. Intuitively, Γ can be thought of as the stack of binders under which a program evaluation takes place. Δ, on the other hand only contains arguments. If we were to mimic standard induction elimination, Γ were to stay empty since evaluation under λ-binders is not permitted.

For $\lambda^{\Rightarrow\star\Box}$, we maintain the invariant, that every declaration in Γ is accounted for in the world w.

Definition 8 (Well-worlded context). *We say that a context Γ is well-worlded under w if the judgment $\vdash \Gamma \in w$ is derivable. The rules defining this judgment are given in Figure 2.*

The two right most zones Φ and Ψ belong to $\lambda^{\Rightarrow\star\Box}$. Φ is the standard context one would expect, and Ψ is a context that only contains variable declarations of closed typed type. Φ captures all local arguments, whereas Ψ contains all other functions, that may be called from within an expression.

We begin now with the discussion of the term level of $\lambda^{\Rightarrow\star\Box}$, and simultaneously its static semantics. We write e for expressions, and c for closed expressions. For $\lambda^{\Rightarrow\star\Box}$'s typing judgment for expressions, we write

$$\Gamma; \Delta; \Phi; \Psi \vdash_w e : \tau$$

and for closed expressions

$$\Psi \vdash c \in \sigma$$

4.1 The Simply-Typed Fragment of $\lambda^{\Rightarrow\star\Box}$

$$\text{Expressions } e ::= x \mid \Lambda x : \tau.\, e \mid e_1\ e_2$$

$$\frac{\Phi(x) = \tau}{\Gamma; \Delta; \Phi; \Psi \vdash_w x : \tau}\ \mathsf{ax}$$

$$\frac{\Gamma; \Delta; \Phi, x : \tau; \Psi \vdash_w e : \tau'}{\Gamma; \Delta; \Phi; \Psi \vdash_w \Lambda x : \tau.\, e : \tau \Rightarrow \tau'}\ \mathsf{fun}$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \tau \Rightarrow \tau' \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \tau}{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1\ e_2 : \tau'}\ \mathsf{@}$$

The expressions and the typing rules for this fragment are standard. Note, in the interest of clarity, we use an uppercase Λ-symbol to denote functional terms. Here are the expression for constructing and projecting from pairs.

$$\text{Expressions } e ::= \ldots \mid \langle\!\langle e_1; e_2 \rangle\!\rangle \mid 1(e) \mid 2(e)$$

1 and 2 are projections that bind tighter than any other connective.

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \tau_1 \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \tau_2}{\Gamma; \Delta; \Phi; \Psi \vdash_w \langle\!\langle e_1; e_2 \rangle\!\rangle : \tau_1 \star \tau_2}\ \mathsf{Pair}$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e : \tau_1 \star \tau_2}{\Gamma; \Delta; \Phi; \Psi \vdash_w 1(e) : \tau_1}\ \mathsf{1} \qquad \frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e : \tau_1 \star \tau_2}{\Gamma; \Delta; \Phi; \Psi \vdash_w 2(e) : \tau_2}\ \mathsf{2}$$

4.2 The Modal Fragment of $\lambda^{\Rightarrow\star\Box}$

$$\text{Expressions } e ::= \ldots \mid \mathsf{box}\ M \mid \text{let } \mathsf{box}\ x = e_1 \text{ in } e_2 : \tau$$

$$\frac{\Gamma, \Delta \vdash_w M : A}{\Gamma; \Delta; \Phi; \Psi \vdash_w \mathsf{box}\ M : \Box A}\ \mathsf{box}$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash e_1 : \Box A \qquad \Gamma; \Delta, x : A; \Phi; \Psi \vdash e_2 : \tau}{\Gamma; \Delta; \Phi; \Psi \vdash_w \text{let } \mathsf{box}\ x = e_1 \text{ in } e_2 : \tau}\ \mathsf{letbox}$$

The two expressions are standard as well [15]. What is less standard, however, is the $\lambda^{\rightarrow\times}$ typing judgment in the premiss of the introduction rule box. M is valid under the union of Γ and Δ, but may not depend on Φ nor Ψ. The letbox rule allows variables to move from the $\lambda^{\Rightarrow\star\Box}$ context Ψ into the $\lambda^{\rightarrow\times}$ context Δ.

4.3 Homomorphic Image of $\lambda^{\rightarrow\times}$

$$\begin{aligned}\text{Expressions } e ::= {} & \ldots \mid [\lambda]x : \Box A.\, e \mid e_1[\cdot]e_2 \\ & \mid [\langle]e_1; e_2[\rangle] \mid [\pi_1]\ e \mid [\pi_2]\ e\end{aligned}$$

$$\frac{\vdash A \in w \qquad \Gamma, x : A; \Delta; \Phi; \Psi \vdash_w e : \Box B}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\lambda]x : A.\, e : \Box(A \rightarrow B)}\ [\mathsf{lam}]$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \Box B \rightarrow A \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \Box B}{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1[\cdot]e_2 : \Box A}\ [\mathsf{app}]$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \Box A \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \Box B}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\langle]e_1, e_2[\rangle] : \Box(A \times B)}\ [\mathsf{pair}]$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e : \Box(A \times B)}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\pi_1]\ e : \Box A}\ [\mathsf{fst}]$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e : \Box(A \times B)}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\pi_2]\ e : \Box B}\ [\mathsf{snd}]$$

The five expressions are the homomorphic image of the object constructors on the $\lambda^{\rightarrow\times}$ level. Consequently the five defining rules are the homomorphic image of the $\lambda^{\rightarrow\times}$ five $\lambda^{\rightarrow\times}$ rules depicted in Figure 1. This similarity is not accidental. For example in order to construct a pair in $\lambda^{\Rightarrow\star\Box}$, we will use the $[\langle]\cdot;\cdot[\rangle]$ term, and similarly $[\pi_i]$ for projections, $[\cdot]$ for applications, and $[\lambda]$ for higher-order abstraction. The most interesting rule is of course [lam]. The rule permits evaluation under any $x : A$, as long as adding A to context Γ does not render it ill-worlded. Recall that Γ stands for the stack of λ-binders under which evaluation is to take place.

4.4 Case Analysis

$$\text{Expressions } e ::= \ldots \mid \P x : A.\, e \mid \exists x : A.\, e \mid e \mapsto e'$$

$$\frac{\vdash A \in w \qquad \Gamma, x : A; \Delta; \Phi; \Psi \vdash_w e : \tau'}{\Gamma; \Delta; \Phi; \Psi \vdash_w \P x : A.\, e : \tau'}\ \mathsf{ex}\Gamma$$

$$\frac{\Gamma; \Delta, x : A; \Phi; \Psi \vdash_w e : \tau'}{\Gamma; \Delta; \Phi; \Psi \vdash_w \exists x : A.\, e : \tau'}\ \mathsf{ex}\Delta$$

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \Box A \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \tau}{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 \mapsto e_2 : \Box A \Rightarrow \tau}\ \mathsf{case}$$

$\lambda^{\Rightarrow\star\Box}$ can not only construct $\lambda^{\rightarrow\times}$-objects, but can also take them apart. A case expression must be of type $\Box A \Rightarrow \tau$, which is reminiscent of $\Box A \rightarrow B$ in the modal λ-calculus [15]. A case has the form $e_1 \mapsto e_2$ and is usually prefixed by a list of existential quantifiers, that bind all free variables in the pattern e_1. It is important to note that there are two different kinds of pattern variables. The quantifier ¶ ranges over universally quantified parameters that can only be instantiated by another $x \in \Gamma$. The quantifier $\exists^\Delta$ on the other hand can be matched by any $\lambda^{\rightarrow\times}$ object. Therefore, in the premiss of rule $\mathsf{ex}\Gamma$ ($\mathsf{ex}\Delta$), $x : A$ is declared in Γ (Δ). Note, that in $\lambda^{\Rightarrow\star\Box}$, any object of type $\Box A$ may serve as a pattern. This is possible because any e of type $\Box A$ will evaluate to a box M as we will show in Theorem 4.

4.5 Non-Determinism

$$\text{Expressions } e ::= \ldots \mid (e_1 | e_2)$$

$$\frac{w_1 \leq w \quad w_2 \leq w}{w_1 + w_2 \leq w}\ \textsf{sub\_alt} \qquad \frac{\vdash A \in w}{A \leq w}\ \textsf{sub\_atom}$$

$$\frac{}{\cdot \leq w}\ \textsf{sub\_empty}$$

Figure 3: World subsumption.

$$\frac{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 : \tau \qquad \Gamma; \Delta; \Phi; \Psi \vdash_w e_2 : \tau}{\Gamma; \Delta; \Phi; \Psi \vdash_w e_1 | e_2 : \tau}\ \textsf{alt}$$

Pattern-matching alone is insufficient unless it is possible to account for failure. To this end, $\lambda^{\Rightarrow\star\Box}$ provides the usual non-deterministic choice operator.

4.6 Recursion

Expressions	$e ::= \ldots \mid \textsf{open}\ c$
Closed Expressions	$c ::= u \mid \textsf{close}\ \langle w; e\rangle \mid \mu u \in \sigma.\, c$

$$\frac{\Psi \vdash c \in \langle w'; \tau\rangle \qquad w' \leq w \qquad w \leq w'}{\Gamma; \Delta; \Phi; \Psi \vdash_w \textsf{open}\ c : \tau}\ \textsf{open}$$

$$\frac{\Psi(u) = \sigma}{\Psi \vdash u \in \sigma}\ \textsf{var}$$

$$\frac{\cdot; \cdot; \cdot; \Psi \vdash_w e : \tau}{\Psi \vdash \textsf{close}\ \langle w; e\rangle \in \langle w; \tau\rangle}\ \textsf{close} \qquad \frac{\Psi, u \in \sigma \vdash e \in \sigma}{\Psi \vdash \mu u \in \sigma.\, e \in \sigma}\ \textsf{rec}$$

A function definition is complete, only if it is closed with respect to the world in which it can be executed. All assumptions of this kind are collected in the context Ψ. There are two possibilities to add a fixed point operator to $\lambda^{\Rightarrow\star\Box}$. One is as an expression, the other as a closed expression. The reason why we have decided to add it as a closed expression is that it is treated exactly the same as a call to an already defined function in Ψ. In both cases, the function must be opened and the world contained within checked. To be safe, the worlds of the caller and the callee must mutually be subsume each other, which is expressed by the premisses $w' \leq w$ and $w \leq w'$ in rule open. The rules defining *world subsumption* are given in Figure 3. With further refined analysis on c, this condition may be relaxed.

A $\lambda^{\Rightarrow\star\Box}$ expression e can only be closed by `close`, if does not contain any free $\lambda^{\rightarrow\times}$ variables, or $\lambda^{\Rightarrow\star\Box}$ variables. This restriction is necessary for functions not to escape their world. Recursion looks, perhaps, surprisingly straightforward.

5. SEMANTICS

The modal operator $\Box A$ embeds a strongly normalizing calculus $\lambda^{\rightarrow\times}$, where every object is equivalent to a canonical form, into a non-terminating calculus $\lambda^{\Rightarrow\star\Box}$, whose functions may be defined by cases, and general recursion. In this section we discuss the operational meaning of $\lambda^{\Rightarrow\star\Box}$ that assigns a semantic value to each well-typed (closed) expression. Ill-typed terms in $\lambda^{\Rightarrow\star\Box}$ are semantically undefined.

Definition 9 (Values).

Values	$v ::= \textsf{box}\ M \mid \Lambda x : \tau.\, e \mid e_1 \mapsto e_2$
Closed Values	$z ::= \textsf{close}\ e$

The domain of observable values of $\lambda^{\Rightarrow\star\Box}$ are functions, and objects of modal type. In Section 8 we show, that any well-typed term in $\lambda^{\Rightarrow\star\Box}$ is semantically valid. The operational semantics of $\lambda^{\Rightarrow\star\Box}$ is specified as a big-step semantics, alternatively, a small-step semantics would have worked as well. By its very nature, the big-step semantics imposes an evaluation order on subgoals.

In the interest of clarity we have left the operational semantics so general, that desired features, such as determinism, progress, termination, are not enforced, but left to the programmer. In our system, for example, it possible to write a non-deterministic program that apparently evaluates to any value

$$\exists x : A.\, \textsf{box}\ x.$$

This is a feature, and not a bug. It is possible to devise restricted type systems, that enforce determinism, coverage, and termination up to a certain degree, but we feel that this discussion is beyond the scope of this paper and will be studied in future work. For example, to guarantee determinism, we could strengthen the type system and only allow the existential quantifiers ¶ and $\exists^{\Delta}$ to occur as prefixes to patterns in which all variables are bound in positions, where matching is guaranteed to instantiate them. As a refinement we could permit $¶x : A.\, e$ to exists anywhere inside a term e, and employ a backtracking operational semantics to select among the finitely many possibilities a suitable instantiation for the $x : A$.

We begin now with the definition of $\lambda^{\Rightarrow\star\Box}$'s operational semantics that is captured by two evaluation judgments

$$\Gamma \vdash_w e \hookrightarrow v$$

for expressions, and

$$\vdash c \hookrightarrow z$$

for closed expressions.

5.1 The Simply-Typed Fragment of $\lambda^{\Rightarrow\star\Box}$

$$\frac{}{\Gamma \vdash_w \Lambda x : \tau.\, e \hookrightarrow \Lambda x : \tau.\, e}\ \textsf{ev}\Lambda$$

$$\frac{\begin{array}{l}\Gamma \vdash_w e_1 \hookrightarrow (\Lambda x : \tau.\, e_1') \\ \Gamma \vdash_w e_2 \hookrightarrow v_2 \qquad\qquad \Gamma \vdash_w [v_2/x]e_1' \hookrightarrow v\end{array}}{\Gamma \vdash_w e_1\ e_2 \hookrightarrow v}\ \textsf{strict}$$

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow (\Lambda x : \tau.\, e_1') \qquad \Gamma \vdash_w [e_2/x]e_1' \hookrightarrow v}{\Gamma \vdash_w e_1\ e_2 \hookrightarrow v}\ \textsf{lazy}$$

The first set of evaluation rules is straightforward. We have given both, a rule for strict evaluation and an alternative rule for lazy evaluation.

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow v_1 \qquad \Gamma \vdash_w e_2 \hookrightarrow v_2}{\Gamma \vdash_w \langle\!\langle e_1; e_2\rangle\!\rangle \hookrightarrow \langle\!\langle v_1; v_2\rangle\!\rangle}\ \textsf{ev}\langle\!\langle\rangle\!\rangle$$

$$\frac{\Gamma \vdash_w e \hookrightarrow \langle\!\langle v_1; v_2\rangle\!\rangle}{\Gamma \vdash_w 1(e) \hookrightarrow v_1}\ \textsf{ev1} \qquad \frac{\Gamma \vdash_w e \hookrightarrow \langle\!\langle v_1; v_2\rangle\!\rangle}{\Gamma \vdash_w 2(e) \hookrightarrow v_2}\ \textsf{ev2}$$

5.2 The Modal Fragment of $\lambda^{\Rightarrow\star\Box}$

$$\frac{}{\Gamma \vdash_w \mathsf{box}\ M \hookrightarrow \mathsf{box}\ M}\ \mathsf{evbox}$$

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow \mathsf{box}\ M \qquad \Gamma \vdash_w [M/x]e_2 \hookrightarrow v}{\Gamma \vdash_w \mathrm{let}\ \mathsf{box}\ x = e_1\ \mathrm{in}\ e_2 \hookrightarrow v}\ \mathsf{evletbox}$$

Those two rules demonstrate how information is exchanged between $\lambda^{\rightarrow\times}$ and $\lambda^{\Rightarrow\star\Box}$. A boxed expression is value and hence only evaluates to itself. The second rule illustrates how $\lambda^{\rightarrow\times}$ objects, whose existence is proclaimed by the $\lambda^{\Rightarrow\star\Box}$ type system is captured by opening the box, and substituted into the body of the let expression.

5.3 Homomorphic Image of $\lambda^{\rightarrow\times}$

$$\frac{\Gamma, x : A \vdash_w e \hookrightarrow \mathsf{box}\ M}{\Gamma \vdash_w [\lambda]x : A.\, e \hookrightarrow \mathsf{box}\ (\lambda x : A.\, M)}\ \mathsf{ev}[\lambda]$$

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow \mathsf{box}\ (\lambda x : A.\, M_1) \qquad \Gamma \vdash_w e_2 \hookrightarrow \mathsf{box}\ M_2}{\Gamma \vdash_w e_1[\cdot]e_2 \hookrightarrow \mathsf{box}\ ([M_2/x]M_1)}\ \mathsf{ev}[\cdot]$$

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow \mathsf{box}\ M_1 \qquad \Gamma \vdash_w e_2 \hookrightarrow \mathsf{box}\ M_2}{\Gamma \vdash_w [\langle]e_1, e_2[\rangle] \hookrightarrow \mathsf{box}\ \langle M_1; M_2\rangle}\ \mathsf{ev}[\langle][\rangle]$$

$$\frac{\Gamma \vdash_w e \hookrightarrow \mathsf{box}\ \langle M_1; M_2\rangle}{\Gamma \vdash_w [\pi_1]\ e \hookrightarrow \mathsf{box}\ M_1}\ \mathsf{ev}[\pi_1]$$

$$\frac{\Gamma \vdash_w e \hookrightarrow \mathsf{box}\ \langle M_1; M_2\rangle}{\Gamma \vdash_w [\pi_2]\ e \hookrightarrow \mathsf{box}\ M_2}\ \mathsf{ev}[\pi_2]$$

In the homomorphic image, every single rule evaluates its respective arguments to boxed $\lambda^{\rightarrow\times}$ objects, open these boxes, and recombines the result accordingly. The only rule that does anything interesting is thus the $\mathsf{ev}[\lambda]$ rule because it allows evaluation under functional abstraction. $x : A$ is pushed onto the stack of open parameters Γ, and the evaluation continues. By construction, it is never possible to consider cases on $x : A$, and thus x can only occur as a parameter inside $\mathsf{box}\ M$. Consequently, by rule lam the term $\mathsf{box}\ (\lambda x : A.\, M)$ is valid.

5.4 Case Analysis

$$\frac{\Gamma \vdash_w M : A \qquad \Gamma \vdash_w [M/x]e \hookrightarrow v}{\Gamma \vdash_w \exists x : A.\, e \hookrightarrow v}\ \mathsf{inst}\Delta$$

$$\frac{\Gamma(y) = A \qquad \Gamma \vdash_w [y/x]e \hookrightarrow v}{\Gamma \vdash_w \P x : A.\, e \hookrightarrow v}\ \mathsf{inst}\Gamma$$

$$\frac{}{\Gamma \vdash_w (e_1 \mapsto e_2) \hookrightarrow (e_1 \mapsto e_2)}\ \mathsf{evcase}$$

$$\frac{\begin{array}{l}\Gamma \vdash_w e_1 \hookrightarrow ([M_1] \mapsto e_1') \\ \Gamma \vdash_w e_2 \hookrightarrow [M_2] \qquad\qquad M_1 \equiv M_2 \qquad \Gamma \vdash_w e_1' \hookrightarrow v\end{array}}{\Gamma \vdash_w e_1\ e_2 \hookrightarrow v}\ \mathsf{match}$$

That $\lambda^{\Rightarrow\star\Box}$ supports case analysis on any functional term of $\lambda^{\rightarrow\times}$ is arguable one of its most distinguished features. The first two rules (non-deterministically) instantiate the existentially quantified variables. In $\mathsf{inst}\Delta$ every well-typed term will do, in $\mathsf{inst}\Gamma$, however, the term must be a parameter, declared in Γ. This rules lends Γ the character of a parameter context because the only way to instantiate parameters is by other parameters.

Conversely, allowing x to be instantiated with any well-typed term in $\mathsf{inst}\Gamma$ renders $\lambda^{\Rightarrow\star\Box}$ unsound. To see what goes wrong, consider the following counterexample that takes place in world $\mathsf{Circuit}$.

$$\begin{array}{ll}\multicolumn{2}{l}{\P x : \mathsf{circuit}.} \\ \mathsf{box}\ x \mapsto & (\P o : \mathsf{Circuit}.\, \exists y_1 : \mathsf{circuit}.\, \exists y_2 : \mathsf{circuit}. \\ & (\mathsf{box}\ (o.\mathsf{connect1}\ y_1\ y_2) \mapsto \mathsf{box}\ y_1)\ \mathsf{box}\ x\end{array}$$

evaluates to M_2 if we instantiate x by $(o.\mathsf{connect1}\ M_1\ M_2)$ for a given $o : \mathsf{Circuit}$ $(\in \Gamma)$, because the embedded case analysis succeeds. But case analysis over parameters must be disallowed. This behavior is prevented if we require x to be instantiated by a parameter because the evaluation of the "counter example" necessarily gets stuck.

A case $e_1 \mapsto e_2$ is a value and does not reduce any further. Note that the left argument in an application can evaluate either to $\Lambda x : \tau.\, e$ or to a case $e_1 \mapsto e_1$.

5.5 Non-Determinism

$$\frac{\Gamma \vdash_w e_1 \hookrightarrow v}{\Gamma \vdash_w e_1 \mid e_2 \hookrightarrow v}\ \mathsf{evleft} \qquad \frac{\Gamma \vdash_w e_2 \hookrightarrow v}{\Gamma \vdash_w e_1 \mid e_2 \hookrightarrow v}\ \mathsf{evright}$$

The operational semantics of non-deterministic choice is standard and so are the possible design decisions that an implementor of $\lambda^{\Rightarrow\star\Box}$ has to face in conjunction with deciding how to instantiate the existential choices. Possible options include left to right or right to left evaluation strategies, but includes also back-tracking and the computation of sets of values.

5.6 Recursion

$$\frac{\vdash e \hookrightarrow \mathsf{close}\ \langle w'; e\rangle \qquad \Gamma \vdash_{w'} e \hookrightarrow v}{\Gamma \vdash_w \mathsf{open}\ e \hookrightarrow v}\ \mathsf{evopen}$$

$$\frac{}{\vdash \mathsf{close}\ \langle w; e\rangle \hookrightarrow \mathsf{close}\ \langle w; e\rangle}\ \mathsf{evclose}$$

$$\frac{\vdash [\mu u \in \sigma.\, e/u]e \hookrightarrow z}{\vdash \mu u \in \sigma.\, e \hookrightarrow z}\ \mathsf{evrec}$$

The final three rules explain the evaluation of the fixed point, how to open a closed expression, and how to close an open expression. A well-typed program is automatically checked for world subsumption. Thus, a closed program can be opened without a renewed check of the world relation.

This concludes the set of rules that describe the operational semantics of $\lambda^{\Rightarrow\star\Box}$.

6. PSEUDO-η EXPANSION

Many interesting functions can be elegantly written in $\lambda^{\Rightarrow\star\Box}$. Consider, for example, addition on natural numbers.

Example 10 (Addition). *Let* nat *be the type of natural numbers and*

$$\mathsf{Nat} \;=\; \mathsf{nat} \times (\mathsf{nat} \to \mathsf{nat}).$$

the type for its constructors. Zero and successor are the usual constructors, and they are defined as projections from $o : \mathsf{Nat}$.

$$\begin{array}{rcl}\mathsf{z} &=& \lambda o : \mathsf{Nat}.\, \pi_1\ o\\ \mathsf{s} &=& \lambda o : \mathsf{Nat}.\, \pi_2\ o.\end{array}$$

In $\lambda^{\Rightarrow\star\Box}$*, the addition of two numbers can be easily recursively defined. Quite unsurprisingly, no higher-order abstract syntax was used for defining* nat*, and thus, addition is valid in the world* $w_0 = \mathsf{Nat}$ *that only contains the constructors for natural numbers. Recall that we write* $o.\mathsf{z}$ *for* $(\mathsf{z}\ o)$.

$$\begin{array}{rcl} f_+ &=& \mu u \in \langle w_0; \Box(\mathsf{nat}\times\mathsf{nat}) \Rightarrow \Box\mathsf{nat}\rangle.\\ && \quad (\P o : \mathsf{Nat}.\, \exists n : \mathsf{nat}.\, \mathsf{box}\ \langle o.\mathsf{z}; n\rangle \mapsto \mathsf{box}\ n)\\ && \mid\ (\P o : \mathsf{Nat}.\, \exists n_1 : \mathsf{nat}.\, \exists n_2 : \mathsf{nat}.\, \mathsf{box}\ \langle o.\mathsf{s}\cdot n_1; n_2\rangle\\ && \quad \mapsto\ (\mathsf{box}\ (o.\mathsf{s}))[\cdot](\mathsf{open}\ u\ \langle n_1; n_2\rangle))\end{array}$$

On the other hand, there are examples that are not (yet) expressible in the current version of $\lambda^{\Rightarrow\star\Box}$, such as a function that counts all variables that occur in a λ-term. The problem itself is not grave, but instructional. Therefore we postponed the discussion of the problem and its solution until now.

Example 11 (Counting variables in λ-terms.). *Let* exp *be a type that ranges over λ-expressions and*

$$\begin{array}{rcl}\mathsf{Exp} &=& ((\mathsf{exp} \to \mathsf{exp}) \to \mathsf{exp})\\ && \times\ (\mathsf{exp} \to (\mathsf{exp} \to \mathsf{exp}))\end{array}$$

a type that defines the usual two constructors for λ-terms in higher-order abstract syntax.

$$\begin{array}{rcl}\mathsf{lam} &=& \lambda o : \mathsf{Exp}.\, \pi_1\ o\\ \mathsf{app} &=& \lambda o : \mathsf{Exp}.\, \pi_2\ o.\end{array}$$

The following $\lambda^{\Rightarrow\star\Box}$ *function* f_{ct} *is designed to count the number of variables that occur in a term. Although it looks correct, it is ill-typed in* $\lambda^{\Rightarrow\star\Box}$ *as we discuss next.*

$$\begin{array}{rcl} w_1 &=& \mathsf{Exp} + \mathsf{exp}\\ f_{ct} &=& \mu u \in \langle w_1; \Box\mathsf{exp} \Rightarrow \Box\mathsf{nat}\rangle.\\ && \quad (\P x : \mathsf{exp}.\, \P n : \mathsf{Nat}.\, \mathsf{box}\ x \mapsto \mathsf{box}\ (n.\mathsf{s}\cdot n.\mathsf{z}))\\ && \mid\ (\P o : \mathsf{Exp}.\, \exists e : (\mathsf{exp}\to\mathsf{exp}).\, \mathsf{box}\ (o.\mathsf{lam}\ e)\\ && \quad\ \mapsto\ ([\lambda]x : \mathsf{exp}.\, (\mathsf{open}\ u\ ((\mathsf{box}\ e)[\cdot](\mathsf{box}\ x)))))\\ && \mid\ (\P o : \mathsf{Exp}.\, \exists e_1 : \mathsf{exp}.\, \exists e_2 : \mathsf{exp}.\, \mathsf{box}\ (o.\mathsf{app}\ e_1\ e_2)\\ && \quad\ \mapsto\ (\mathsf{open}\ f_+\ [\langle]\mathsf{open}\ u\ (\mathsf{box}\ e_1); \mathsf{open}\ u\ (\mathsf{box}\ e_2)[\rangle]))\end{array}$$

The problem of f_{ct} lies in use of $[\lambda]$. The typing rule [lam] asserts that the result type of

$$[\lambda]x : \mathsf{exp}.\, (\mathsf{open}\ u\ ((\mathsf{box}\ e)[\cdot](\mathsf{box}\ x))) \tag{3}$$

is $\Box(\mathsf{exp} \to \mathsf{nat})$. However, the the type of f_{ct} is $\Box\mathsf{nat}$. Therefore the second case of f_{ct} causes a type error.

But note, that the no canonical form of type nat will ever contain a subexpression of type exp. Therefore, (3) is guaranteed to evaluate to a constant function of type $\Box(\mathsf{exp} \to \mathsf{nat})$, rendering the preceeding functional abstraction unnecessary. The current version of the rule [lam] is thus too rigid.

$$\frac{A_1 \in w \qquad \Gamma, x : A; \Delta; \Phi; \Psi \vdash_w e : \Box B}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\lambda]x : A.\, e : \Box(A \to B)}\ [\mathsf{lam}]$$

The world system dictates the exact form of hypotheses $x : A$. Using projections, a programmer can select constructors that go into a recursive call, but with the current formulation of the rule, he or she cannot control which projections to abstract.

One could try to solve this problem on the $\lambda^{\Rightarrow\star\Box}$ level. After all $[\lambda]$ is the homomorphic image of $\lambda^{\to\times}$ abstraction. However, one discovers fast, that breaking up a composite object has consequences for the world the $\lambda^{\Rightarrow\star\Box}$ function is defined in, which in turn adds additional cases to the function itself and is therefore doomed to failure.

An alternative solution to this problem and the one we favor is to alter the $[\lambda]$ rule in such a way, that the user gives a "pseudo" η-expanded form of the function. In η-long form a functional term e has the form $\lambda x : A.\, e\ x$, and not just $\lambda x : A.\, e$. The argument x to e captures the fact that e takes one argument of type x. In our situation, we will write $[\lambda]x : A.\, e_1[\cdot]e_2[\cdot]\ldots[\cdot]e_n$. to signal that e_1 is a function that may depend on $n-1$ arguments. Of course each e_i must be of type $\Box A_i$ and can therefore not be variables per se but $\lambda^{\to\times}$ expressions, such as $(\pi_1\ x)$, or $(\pi_2\ (\pi_2\ (\pi_2\ x)))$. For Example 11 that opened up this discussion, this implies that under this new interpretation f_{ct} is type correct, because the offending return object is rightly $\Box\mathsf{nat}$. The "pseudo" η-long form of type $\Box(\mathsf{exp} \to \mathsf{nat})$ is $[\lambda]x : \mathsf{exp}.\, (\mathsf{open}\ u\ ((\mathsf{box}\ e)[\cdot](\mathsf{box}\ x)))[\cdot](\mathsf{box}\ x)$.

We proceed with modifying the typing and evaluation rule to accommodate "pseudo" η-long forms.

$$\frac{A \in w \qquad \Gamma, x : A; \Delta; \Phi; \Psi \vdash_w e_i : \Box A_i}{\Gamma; \Delta; \Phi; \Psi \vdash_w [\lambda]x : A.\, e_1[\cdot]e_2[\cdot]\ldots[\cdot]e_n : \Box(A_2 \to \ldots \to A_n \to A_1)}\ [\mathsf{lam}]$$

The important question to settle next is, of course, how to evaluate a $[\lambda]$ expression.

$$\frac{\Gamma, x : A \vdash_w e_i \hookrightarrow \mathsf{box}\ M_i \qquad M = M_1[id_\Gamma, M_2/x_2, \ldots, M_n/x_n]^{-1}}{\Gamma \vdash_w [\lambda]x : A.\, e_1[\cdot]e_2[\cdot]\ldots[\cdot]e_n \hookrightarrow \mathsf{box}\ (\lambda x_2 : A_2.\, \ldots \lambda x_n : A_n.\, M)}\ \mathsf{ev}[\lambda]$$

We write id_Γ for the simultaneous identity substitution on Γ. We proceed by evaluating the body of the term e_1 in an extended context $\Gamma, x : A$, and subsequently the remaining expressions $e_2 \ldots e_n$ that are intended to correspond to the different variables of our "pseudo"η-long form. As part of the result of the overall evaluation, we expect to return

$$\Gamma, x_2 : A_2, \ldots, x_n : A_n \vdash_w M : A_1,$$

that under the simultaneous substitution

$$\sigma = (id_\Gamma, M_1/x_1, \ldots, M_n/x_n)$$

should be equivalent to M', the result of evaluation e_1:

$$M[id_\Gamma, M_1/x_1, \ldots, M_n/x_n] = M'.$$

Such a term M may, of course not always exists, because not every term is in the image of substitution application. And if it exists, it need not be unique, without side conditions attached to the argument terms.

Often, however a unique M term exists, especially, if the arguments are fully expanded non-overlapping sequences of projections. To compute M from M', we apply the technique of inverse substitutions that has been studied for explicity pattern substitution calculi [5]. Following an algorithm outlined there, we define $M[\sigma]^{-1}$ by the following rules.

$$
\begin{array}{lcl}
M[\sigma]^{-1} & = & x \quad \text{if } \sigma(x) = M \text{ for some } x \\
(M_1 \cdot M_2)[\sigma]^{-1} & = & (M_1[\sigma]^{-1}) \cdot (M_2[\sigma]^{-1}) \\
(\lambda x : A.\, M)[\sigma]^{-1} & = & (\lambda x : A.\, M[\sigma, x/x]^{-1}) \\
\langle M_1; M_2\rangle[\sigma]^{-1} & = & \langle M_1[\sigma]^{-1}; M_2[\sigma]^{-1}\rangle \\
(\pi_1\ M)[\sigma]^{-1} & = & \pi_1\ (M[\sigma]^{-1}) \\
(\pi_2\ M)[\sigma]^{-1} & = & \pi_2\ (M[\sigma]^{-1}) \\
(M)[\sigma]^{-1} & = & \text{undefined otherwise}
\end{array}
$$

Fully expanded sequences of projections play the role of variables, and thus the properties of the inversion of pattern substitutions directly applies to our setting. If the arguments in the "pseudo" η-long form are all pairwise different parameters, and include all parameters that can potentially occur inside the result objects, the premiss of the $\mathsf{ev}[\lambda]$-rule $M = M_1[id_\Gamma, M_2/x_2, \ldots, M_n/x_n]^{-1}$ will always succeed.

However, if not used with care, inversion will either fail (which means the evaluation gets stuck), or may return several solutions. We leave it to future work to develop criteria that decide if inversion will result in a unique answer.

7. EXAMPLES

We illustrate the use of $\lambda^{\rightarrow\times}$ and $\lambda^{\Rightarrow\star\Box}$ in terms of a few more examples.

Example 12 (Circuits). *Returning to Example 1 we give as one representative a function f_c that replaces all $\wedge$ and $\neg$ gates in a circuit by $\overline{\wedge}$ gates. The function lives in world $w_1 = \mathsf{Circuit} + \mathsf{circuit}$.*

$$
\begin{array}{lll}
f_c & = & \mu u \in \langle w_1; \Box\mathsf{circuit} \Rightarrow \Box\mathsf{circuit}\rangle. \\
 & & \quad (\P o : \mathsf{circuit}.\, \mathsf{box}\ o \mapsto \mathsf{box}\ o) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \exists x_1 : \mathsf{circuit}.\, \exists x_2 : \mathsf{circuit}. \\
 & & \qquad\qquad \mathsf{box}\ (o.\mathsf{connect1} \cdot x_1 \cdot x_2) \\
 & & \quad\ \mapsto\ (\mathsf{box}\ (o.\mathsf{connect1})) \\
 & & \qquad\qquad [\cdot](\mathsf{open}\ u\ (\mathsf{box}\ x_1)) \\
 & & \qquad\qquad [\cdot](\mathsf{open}\ u\ (\mathsf{box}\ x_2)))) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \exists x : (\mathsf{circuit} \rightarrow \mathsf{circuit}). \\
 & & \qquad\qquad \mathsf{box}\ (o.\mathsf{connector1} \cdot x) \\
 & & \quad\ \mapsto\ (\mathsf{box}\ (o.\mathsf{connector1})) \\
 & & \qquad\qquad [\cdot]([\lambda]o' : \mathsf{circuit}.\, (\mathsf{open}\ u \\
 & & \qquad\qquad\quad ((\mathsf{box}\ x)[\cdot](\mathsf{box}\ o')))[\cdot](\mathsf{box}\ x))) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \exists x : \mathsf{circuit}.\, \mathsf{box}\ (o.\mathsf{not} \cdot x) \\
 & & \quad \mapsto\ (\mathsf{box}\ (o.\mathsf{connect1})) \\
 & & \qquad\qquad [\cdot](\mathsf{open}\ u \cdot (\mathsf{box}\ x)) \\
 & & \qquad\qquad [\cdot](\mathsf{box}\ (o.\mathsf{connector1}) \\
 & & \qquad\qquad\quad (\lambda x' : \mathsf{circuit}.\, o.\mathsf{nand} \cdot x' \cdot x'))) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \exists x_1 : \mathsf{circuit}.\, \exists x_2 : \mathsf{circuit}. \\
 & & \qquad\qquad \mathsf{box}\ (o.\mathsf{and} \cdot x_1 \cdot x_2) \\
 & & \quad \mapsto (\mathsf{open}\ u\ (\mathsf{box}\ (o.\mathsf{nand} \cdot (o.\mathsf{not} \cdot x_1) \cdot (o.\mathsf{not} \cdot x_2))))) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \exists x_1 : \mathsf{circuit}.\, \exists x_2 : \mathsf{circuit}. \\
 & & \qquad\qquad \mathsf{box}\ (o.\mathsf{nand} \cdot x_1 \cdot x_2) \\
 & & \quad \mapsto\ (\mathsf{box}\ (o.\mathsf{nand})) \\
 & & \qquad\qquad [\cdot](\mathsf{open}\ u\ (\mathsf{box}\ x_1))) \\
 & & \qquad\qquad [\cdot](\mathsf{open}\ u\ (\mathsf{box}\ x_2))) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \mathsf{box}\ (o.\mathsf{true}) \mapsto \mathsf{box}\ (o.\mathsf{true})) \\
 & & \mid\ (\P o : \mathsf{Circuit}.\, \mathsf{box}\ (o.\mathsf{false}) \mapsto \mathsf{box}\ (o.\mathsf{false})) \\
 & & \mid\ \ldots
\end{array}
$$

The ... stand for the remaining cases connect2, connector2, *and* loop *that follow the pattern of* connect1 *and* connector1.

Example 13 (λ-calculus). *We return to the λ-calculus Example 11, and show how to copy a λ-term into another. To make this example more interesting we change the target type to* exp' *that is defined as follows.*

$$
\begin{array}{lcl}
\mathsf{Exp'} & = & ((\mathsf{exp'} \rightarrow \mathsf{exp'}) \rightarrow \mathsf{exp'}) \\
 & & \times\, (\mathsf{exp'} \rightarrow (\mathsf{exp'} \rightarrow \mathsf{exp'})).
\end{array}
$$

The two constructors are again projections.

$$
\begin{array}{lcl}
\mathsf{lam'} & = & \lambda o : \mathsf{Exp'}.\, \pi_1\ o \\
\mathsf{app'} & = & \lambda o : \mathsf{Exp'}.\, \pi_2\ o.
\end{array}
$$

Besides constructors for λ-terms, the world $w_2 = \mathsf{Exp} + \mathsf{Exp'} + \mathsf{exp} \times \mathsf{exp'}$ may contain pairs of parameters $\mathsf{exp} \times \mathsf{exp'}$. In the definition of the function f_{cp} observe how in the lam *case the parametric function of type $\Box(\mathsf{exp} \rightarrow \mathsf{exp})$ is being translated into a parametric function of type $\Box(\mathsf{exp'} \rightarrow \mathsf{exp'})$.*

$$
\begin{array}{lll}
f_{id} & = & \mu u \in \langle w_2; \Box\mathsf{exp} \Rightarrow \Box\mathsf{exp'}\rangle. \\
 & & \quad (\P x : \mathsf{exp} \times \mathsf{exp'}.\, \mathsf{box}\ (\pi_1\ x) \mapsto \mathsf{box}\ (\pi_2\ x)) \\
 & & \mid\ (\P o : \mathsf{Exp}.\, \exists e : (\mathsf{exp} \rightarrow \mathsf{exp}).\, \mathsf{box}\ (o.\mathsf{lam}\ e) \\
 & & \quad \mapsto\ \mathsf{box}\ (o.\mathsf{lam'})[\cdot] \\
 & & \qquad ([\lambda]x : \mathsf{exp} \times \mathsf{exp'}.\, (\mathsf{open}\ u\ ((\mathsf{box}\ e) \\
 & & \qquad\quad [\cdot](\mathsf{box}\ (\pi_1\ x))))[\cdot](\mathsf{box}\ (\pi_2\ x)))) \\
 & & \mid\ (\P o : \mathsf{Exp}.\, \exists e_1 : \mathsf{exp}.\, \exists e_2 : \mathsf{exp}. \\
 & & \qquad \mathsf{box}\ (o.\mathsf{app}\ e_1\ e_2) \\
 & & \quad \mapsto\ (\mathsf{box}\ (o.\mathsf{app'}))[\cdot](\mathsf{open}\ u\ (\mathsf{box}\ e_1)) \\
 & & \qquad [\cdot](\mathsf{open}\ u\ (\mathsf{box}\ e_2)))
\end{array}
$$

Example 14 (Parallel reduction). *We continue to work with the λ-expressions from Example 11 and give a function that executes parallel reduction on λ-terms. A version of this function implemented in the modal λ-calculus with iterators can be found in [15]. First, we state the function informally:*

$$
\begin{array}{lcl}
\mathsf{par}\ (\mathsf{app}\ e_1\ e_2) & = & \mathsf{par'}\ \langle e_1; \mathsf{par}\ e_2\rangle \\
\mathsf{par}\ (\mathsf{lam}\ e_1) & = & \mathsf{lam}\ (\lambda x : \mathsf{exp}.\, \mathit{new}\ x'.\, \mathsf{par}\ (e_1\ x') \\
 & & \qquad \mathit{where}\ \mathsf{par}\ x' = x \\
 & & \qquad \mathit{and}\ \mathsf{par'}\langle x'; e_3\rangle = \mathsf{app}\ x\ e_3 \\
\mathsf{par'}\ \langle \mathsf{app}\ e_1\ e_2; e_2'\rangle & = & \mathsf{app}\ (\mathsf{par'}\langle e_1; (\mathsf{par}\ e_2)\rangle)\ e_2' \\
\mathsf{par'}\ \langle \mathsf{lam}\ e_1; e_2'\rangle & = & \mathit{new}\ x.\, \mathsf{par}\ (e_1\ x) \\
 & & \qquad \mathit{where}\ \mathsf{par}\ x\ =\ e_2' \qquad (*) \\
 & & \qquad \mathit{and}\ \mathsf{par'}\langle x; e_3\rangle = \mathsf{app}\ e_2'\ e_3
\end{array}
$$

This example is most challenging because it shows the use of mutual recursive functions. In addition, it must distinguish between parameters introduced by par *and parameters introduced by* par'. *At the first glance our world system may not seem general enough to capture this fact, however, it can. First we declare* tag : type *that represents the origin of a parameter.*

$$
\mathsf{Tag} \quad = \quad \mathsf{tag} \times (\mathsf{exp} \rightarrow \mathsf{tag}).
$$

The origins correspond to the two constructors.

$$
\begin{array}{lcl}
\mathsf{par} & = & \lambda t : \mathsf{Tag}.\, \pi_1\ t \\
\mathsf{par'} & = & \lambda t : \mathsf{Tag}.\, \pi_2\ t.
\end{array}
$$

The tag $t.\mathsf{par'}$ has type $\mathsf{exp} \rightarrow \mathsf{tag}$ which we will use to carry the second argument e_2' through a recursive call, to the place where it is needed (see $(*)$). Function f_{pr} encodes parallel reduction and lives in world

$$
w_3 = \mathsf{Exp} + \mathsf{Tag} + ((\mathsf{tag} \rightarrow \mathsf{exp}) \times \mathsf{exp}).
$$

$$
\begin{array}{rcl}
f_{pr} & = & \mu u \in \langle w_3; (\Box\mathsf{exp} \Rightarrow \Box\mathsf{exp}) \star (\Box(\mathsf{exp} \times \mathsf{exp}) \Rightarrow \Box\mathsf{exp})\rangle. \\
& & \langle\!\langle \;\; (\P x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}. \\
& & \qquad \P y : \mathsf{Tag}.\, \mathsf{box}\, ((\pi_1\ x) \cdot y.\mathsf{par}) \\
& & \qquad\quad \mapsto \mathsf{box}\, (\pi_2\ x)) \\
& & \mid \;\; (\P x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}. \\
& & \qquad \P y : \mathsf{Tag}.\, \exists e_2' : \mathsf{exp}.\, \mathsf{box}\, ((\pi_1\ x) \cdot (y.\mathsf{par'} \cdot e_2')) \\
& & \qquad\quad \mapsto \mathsf{box}\, e_2') \\
& & \mid \;\; (\P x : \mathsf{Exp}.\, \exists e_1 : \mathsf{exp}.\, \exists e_2 : \mathsf{exp}.\, \mathsf{box}\, (x.\mathsf{app} \cdot e_1 \cdot e_2) \\
& & \;\; \mapsto \;\; 2(\mathsf{open}\ u)\ \langle \mathsf{box}\ e_1; (1(\mathsf{open}\ u)\ (\mathsf{box}\ e_2))\rangle \\
& & \mid \;\; (\P x : \mathsf{Exp}.\, \exists e_1 : (\mathsf{exp} \to \mathsf{exp}).\, \mathsf{box}\, (x.\mathsf{lam} \cdot e_1) \\
& & \;\; \mapsto \;\; \mathsf{box}\, (o.\mathsf{lam})[\cdot] \\
& & \qquad ([\lambda]x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}.\, ([\lambda]y : \mathsf{Tag}. \\
& & \qquad (1(\mathsf{open}\ u)\ ((\mathsf{box}\ e_1) \\
& & \qquad\quad [\cdot](\mathsf{box}\, ((\pi_1\ x) \cdot y.\mathsf{par})))))[\cdot](\mathsf{box}\, (\pi_2\ x)))) \\
& & ; \;\; (\P x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}.\, \P z : \mathsf{Exp}. \\
& & \qquad \P y : \mathsf{Tag}.\, \exists e_3 : \mathsf{exp}.\, \mathsf{box}\, \langle (\pi_1\ x) \cdot (y.\mathsf{par}); e_3\rangle \\
& & \qquad\quad \mapsto \mathsf{box}\, (z.\mathsf{app} \cdot (\pi_2\ x) \cdot e_3) \\
& & \mid \;\; (\P x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}.\, \P z : \mathsf{Exp}. \\
& & \qquad \P y : \mathsf{Tag}.\, \exists e_2' : \mathsf{exp}.\, \exists e_3 : \mathsf{exp}. \\
& & \qquad\quad \mathsf{box}\, \langle (\pi_1\ x) \cdot (y.\mathsf{par'} \cdot e_2'); e_3\rangle \\
& & \qquad\quad \mapsto \mathsf{box}\, (z.\mathsf{app} \cdot e_2' \cdot e_3)) \\
& & \mid \;\; (\P x : \mathsf{Exp}.\, \exists e_1 : \mathsf{exp}.\, \exists e_2 : \mathsf{exp}.\, \mathsf{box}\, (x.\mathsf{app} \cdot e_1 \cdot e_2) \\
& & \;\; \mapsto \;\; (x.\mathsf{app})[\cdot](2(\mathsf{open}\ u)\ \langle \mathsf{box}\ e_1; \\
& & \qquad\quad (1(\mathsf{open}\ u)\ (\mathsf{box}\ e_2))\rangle)[\cdot](\mathsf{box}\ e_2')) \\
& & \mid \;\; (\P x : \mathsf{Exp}.\, \exists e_1 : (\mathsf{exp} \to \mathsf{exp}).\, \mathsf{box}\, (x.\mathsf{lam} \cdot e_1) \\
& & \;\; \mapsto \;\; [\lambda]x : (\mathsf{tag} \to \mathsf{exp}) \times \mathsf{exp}.\, ([\lambda]y : \mathsf{Tag}. \\
& & \qquad (1(\mathsf{open}\ u)\ ((\mathsf{box}\ e_1) \\
& & \qquad\quad [\cdot](\mathsf{box}\, ((\pi_1\ x) \cdot y.\mathsf{par}))))))\rangle\!\rangle
\end{array}
$$

The only noteworthy difference between the informal and formal implementation of parallel reduction is that the fifth and sixth case of f_{pr} require access to z : Exp *that is not bound in the pattern. However, since we assume that there is only one declaration of* Exp *in the context, non-deterministic choice for z can locate it in one step.*

8. META THEORY

First, we show that the modality $\Box$ fulfills to the very least the distributivity requirement of a modal logic.

LEMMA 15 (DISTRIBUTIVITY). *There exists a function $f : \Box(A \to B) \Rightarrow (\Box A \Rightarrow \Box B)$.*

PROOF. Choose $f = \Lambda x : \Box(A \to B).\, \Lambda y : \Box A.\, x[\cdot]y$. $\Box$

Next, we show value soundness. It might seem to be a trivial property, but it is absolutely essential for soundness of our rules. For example, if $e : \Box A$ reduces to a value, this value should be of the form box M, otherwise the evaluation rules describing the homomorphic image of $\lambda^{\to\times}$ into $\lambda^{\Rightarrow\star\Box}$ can never be applied.

THEOREM 16 (VALUE SOUNDNESS).

1. *If $\mathcal{D}_1 :: \Gamma \vdash_w e \hookrightarrow v$ then v is a value.*
2. *If $\mathcal{D}_2 :: \Gamma \vdash c \hookrightarrow u$ then w is a value.*

PROOF. The proof is by simultaneous structural induction on $\mathcal{D}_1$ and $\mathcal{D}_2$. We don't give the proof here because it follows easily by the inspection of the evaluation rules from Section 5. $\Box$

$\lambda^{\Rightarrow\star\Box}$'s operational semantics ensures that the result of a computation that is begun in a regularly formed world is well-defined in the same world when the computation halts. During evaluation new constructors $x : A$ may be dynamically introduced into Γ, but they are guaranteed to be discharged by the time the computation terminates. $\lambda^{\Rightarrow\star\Box}$'s operational semantics is type preserving. But first we need to show a substitution lemma.

LEMMA 17 (SUBSTITUTION).

1. *If $\Gamma, x : A; \Delta; \Phi; \Psi \vdash_w e : \tau$ and $\Gamma(y) = A$ then*
$$\Gamma; \Delta; \Phi; \Psi \vdash_w e[y/x] : \tau.$$
2. *If $\Gamma; \Delta, x : A; \Phi; \Psi \vdash_w e : \tau$ and $\Gamma \vdash N : A$ then*
$$\Gamma; \Delta; \Phi; \Psi \vdash_w e[N/x] : \tau.$$
3. *If $\Gamma; \Delta; \Phi, x : \tau'; \Psi \vdash_w e : \tau$ and $\Gamma; \Delta; \Phi; \Psi \vdash_w e' : \tau'$ then*
$$\Gamma; \Delta; \Phi; \Psi \vdash_w e[e'/x] : \tau.$$
4. *If $\Gamma; \Delta; \Phi; \Psi, u \in \sigma \vdash_w e : \tau$ and $\Psi \vdash_w c \in \sigma$ then*
$$\Gamma; \Delta; \Phi; \Psi \vdash_w e[c/u] : \tau.$$
5. *If $\Psi, u \in \sigma' \vdash_w c : \sigma$ and $\Psi \vdash_w c' \in \sigma'$ then*
$$\Psi \vdash_w c[c'/u] : \sigma.$$

PROOF. by simultaneous structural induction on the five typing derivations of $e : \tau$ and $c : \sigma$. $\Box$

Considering the complexity of the system, the proofs are all straightforward. For the formulation of the type preservation, we need to introduce a typing relation for substitutions, which we reiterate here. We write $\Gamma' \vdash_w \sigma : \Gamma$ for a simultaneous substitution, that maps variables in x into terms valid in Γ'

$$\frac{\Gamma' \vdash_w \sigma : \Gamma \qquad \Gamma' \vdash_w M : A}{\Gamma' \vdash_w \sigma, M/x : \Gamma, x : A} \qquad \frac{}{\Gamma \vdash_w id_\Gamma : \Gamma}$$

We conclude this section with the type preservation theorem.

THEOREM 18 (TYPE PRESERVATION).

1. *If $\Gamma; \cdot; \cdot; \cdot \vdash_w e : \tau$ and $\Gamma \vdash_w e \hookrightarrow v$ then $\Gamma; \cdot; \cdot; \cdot \vdash_w v : \tau$.*
2. *If $\vdash c \in \sigma$ and $\vdash c \hookrightarrow z$ then $\vdash z \in \sigma$.*
3. *If $\Gamma, x : A \vdash M : A$ and $\Gamma, x_1 : A_1, \ldots x_n : A_n \vdash_w \sigma : \Gamma, x : A$ and $M' = M[\sigma]^{-1}$ (should it exist), then*
$$\Gamma, x_1 : A_1 \ldots x_n : A_n \vdash M' : A.$$

PROOF. by simultaneous structural induction on the three operational meaning, $e \hookrightarrow v$, $c \hookrightarrow w$, and $M' = M[\sigma]^{-1}$. We give one representative case:

Case: *Part 1.* match

$\Gamma \vdash_w e_1\ e_2 \hookrightarrow v$	by assumption
$\Gamma \vdash_w e_1 \hookrightarrow ([M_1] \mapsto e_1')$	by assumption
$\Gamma \vdash_w e_1 \hookrightarrow [M_2]$	by assumption
$M_1 \equiv M_2$	by assumption
$\Gamma \vdash_w e_1' \hookrightarrow v$	by assumption
$\Gamma; \cdot; \cdot; \cdot \vdash_w e_1\ e_2 : \tau$	by assumption
$\Gamma; \cdot; \cdot; \cdot \vdash_w e_1 : \tau' \Rightarrow \tau$	by inversion

$$\begin{array}{ll}
\Gamma; \cdot; \cdot; \cdot \vdash_w e_2 : \tau' & \text{by inversion} \\
\Gamma; \cdot; \cdot; \cdot \vdash_w ([M_1] \mapsto e_1') : \tau' \Rightarrow \tau & \text{by ind. hyp.} \\
\Gamma; \cdot; \cdot; \cdot \vdash_w e_1' : \tau & \text{by inversion} \\
\Gamma; \cdot; \cdot; \cdot \vdash_w v : \tau & \text{by ind. hyp.}
\end{array}$$

$\square$

9. RELATED WORK

Our work was inspired by Miller's system [10], 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 [9] and its refinement by Fegaras and Sheard [6] 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 [6] to circumvent this problem. In contrast to our system, their proposal does not appear to have a logical interpretation.

Closely related to this work is Gabbay and Pitt's work on FreshML [14, 7] an ML-like metalanguage for programming with data structures that may involve variable binding. Like $\lambda^{\Rightarrow\star\square}$, FreshML supports recursive function definitions and pattern matching over its data structures. However, FreshML merely promotes object-level renaming to the meta-level, through a set-theoretic interpretation of name abstraction using FM-sets. Object-level substitution must therefore be implemented for each object language separately. Compared to the $\lambda^{\Rightarrow\star\square}$, FreshML notion of definitional equality is based on α-conversion, whereas $\lambda^{\Rightarrow\star\square}$ permits β- and η-conversion as well. On the other hand the idea of permuting variables, works well with polymorphic datatypes, and it is no clear yet, if and how $\lambda^{\rightarrow\times}$ could be extended to any kind of polymorphism.

In prior work, we have developed the aforementioned modal λ-calculus for combining higher-order abstract syntax and iteration [15]. Both systems attempt to solve the same problem, but on entirely different ways. The modal λ-calculus aims to remain a one-level system, with a modality to separate the parametric function space from the primitive recursive one. The extended system has canonical forms, and is a conservative extension over the simply typed λ-calculus. In the two tiered setting, however, conservative extension becomes less of an issue. Compared to the meta-theory of the modal λ-calculus, the meta theory of $\lambda^{\Rightarrow\star\square}$ is much easier. The existence of canonical forms of $\square A$ is guaranteed by design. Thus, no complex logical relation arguments are necessary, and in fact, all theorems described in this paper have be proved by straightforward simultaneous structural induction. Washburn and Weirich [16] have treated the modal λ-calculus as domain specific language and gave an implementation in Haskell. Using abstract type classes and polymorphism they have shown that programming with higher-order encodings in a simply typed programming language is possible.

10. CONCLUSION

We have presented a λ-calculus for recursive functions that can be defined by pattern-matching over higher-order encodings. It thus complements and refines the logic programming approach to programming with such representations [11, 13].

Our system is satisfactory from the theoretical point of view, can be extended to dependent types and has been implemented in the core of the Delphin programming language. Many natural functions over higher-order representations turn out to be directly definable including one-step parallel reduction.

11. REFERENCES

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

[2] T. Coquand. An algorithm for testing conversion in type theory. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 255–279. Cambridge University Press, 1991.

[3] J. Despeyroux, A. Felty, and A. 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, Apr. 1995. Springer-Verlag LNCS 902.

[4] J. Despeyroux and A. Hirschowitz. Higher-order abstract syntax with induction in Coq. In F. 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.

[5] G. Dowek, T. Hardin, C. Kirchner, and F. Pfenning. Unification via explicit substitutions: The case of higher-order patterns. In M. Maher, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 259–273, Bonn, Germany, Sept. 1996. MIT Press.

[6] L. Fegaras and T. 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, Jan. 1996. ACM Press.

[7] M. Gabbay and A. Pitts. A new approach to abstract syntax involving binders. In G. Longo, editor, *Proceedings of the 14th Annual Symposium on Logic in Computer Science (LICS'99)*, pages 214–224, Trento, Italy, July 1999. IEEE Computer Society Press.

[8] M. J. Gabbay and A. M. Pitts. A new approach to abstract syntax with variable binding. *Formal Aspects of Computing*, 13:341–363, 2001.

[9] E. Meijer and G. 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.

[10] D. 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.

[11] D. 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.

[12] B. Nordström, K. Petersson, and J. 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.

[13] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181. Cambridge University Press, 1991.

[14] A. M. Pitts and M. J. Gabbay. A metalanguage for programming with bound names modulo renaming. In R. Backhouse and J. N. Oliveira, editors, *Mathematics of Program Construction, MPC2000, Proceedings, Ponte de Lima, Portugal, July 2000*, volume 1837 of *Lecture Notes in Computer Science*, pages 230–255. Springer-Verlag, Heidelberg, 2000.

[15] C. Schürmann, J. Despeyroux, and F. Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, (266):1–57, 2001.

[16] G. Washburn and S. Weirich. Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. In *Procoeedings of International Conference on Functional Programming*. ACM Press, 2003. to appear.