Flexible Type Analysis*

Karl Crary
Carnegie Mellon University

Stephanie Weirich
Cornell University

Abstract

Run-time type dispatch enables a variety of advanced optimization techniques for polymorphic languages, including tag-free garbage collection, unboxed function arguments, and flattened data structures. However, modern type-preserving compilers transform types between stages of compilation, making type dispatch prohibitively complex at low levels of typed compilation. It is crucial therefore for type analysis at these low levels to refer to the types of previous stages. Unfortunately, no current intermediate language supports this facility.

To fill this gap, we present the language LX, which provides a rich language of type constructors supporting type analysis (possibly of previous-stage types) as a programming idiom. This language is quite flexible, supporting a variety of other applications such as analysis of quantified types, analysis with incomplete type information, and type classes. We also show that LX is compatible with a type-erasure semantics.

1 Introduction

Type-directed compilers use type information to enable optimizations and transformations that are impossible (or prohibitively difficult) without such information [16, 12, 21, 2, 25, 26, etc.]. However, type-directed compilers for some languages such as Modula-3 and ML face the difficulty that some type information cannot be known at compile time. For example, polymorphic code in ML may operate on inputs of type α where α is not only unknown, but may in fact be instantiated by a variety of different types.

*This material is based on work supported in part by ARPA grant F19628-95-C-C533, AFOSR grant F49620-97-1-0013, and ARPA/RADC grant F30602-1-0317. The second author is also supported by a National Science Foundation Graduate Fellowship.

To appear in the 1999 International Conference on Functional Programming, Paris, France, September 1999.

In order to use type information in contexts where it cannot be provided statically, a number of advanced implementation techniques process type information at run time [12, 21, 30, 23, 26]. Such type information is used in two ways: behind the scenes, typically by tag-free garbage collectors [30, 1], and explicitly in program code, for a variety of purposes such as efficient data representation and marshalling [21, 12, 27]. In this paper we focus on the latter area of applications.

To lay a solid foundation for programs that analyze types at run time, Harper and Morrisett [12] proposed an internal language, called λ_i^{ML}, that supports first-class *intensional analysis* of types (that is, analysis of the structure of types). The λ_i^{ML} language and its derivatives were then used extensively in the high-performance ML compilers TIL/ML [29, 20] and FLINT [27]. The primary novelty of these languages is the presence of "typecase" operators at the level of terms and types, that allow computations and type expressions to depend upon the values of other type expressions at run time.

Like most type-directed compilers, TIL/ML and FLINT preserve types through much of compilation, but discard types at a certain point and finish compilation without them. Nevertheless, there are compelling advantages to preserving types through the entirety of the compiler: types may be used to perform optimizations that are only feasible at low levels, the ability to typecheck intermediate code provides an invaluable tool for debugging a compiler, and types may be used to certify the safety of the output executables [24].

Unfortunately, existing type-analyzing languages are not well suited for further typed compilation. This is because existing such languages are hardwired so that the language's own types are the subject of type analysis. Such a design is quite natural when the language is considered in isolation—what other types are there?—but in the context of a multi-stage, type-directed compiler we face the problem that type-altering transformations (*e.g.,* closure conversion) are applied to intermediate programs that perform type analysis. After such a transformation, we would prefer to preserve the algorithmic structure of our program by continuing to pass and inspect the types that were used before the transformation. Existing type-analyzing languages, which have

```
Λα:Type. λx:α.
  typecase α of
    int                   => ... (* x has type int *) ...
    β × γ                 => ... (* x has type β × γ *) ...
    ∃δ.((δ × β) → γ) × δ  => ... (* x has type ∃δ.((δ × β) → γ) × δ *) ...
    _                     => ... (* x has some other type *) ...
```

Figure 1: Compilation of type analysis, first try

```
Λα:MLType. λx:(interp α).
  typecase α of
    [int]_ML    => ... (* x has type int *) ...
    [β × γ]_ML  => ... (* x has type (interp β) × (interp γ) *) ...
    [β → γ]_ML  => ... (* x has type ∃δ.((δ × (interp β)) → (interp γ)) × δ *) ...
```

Figure 2: Second try

only a single notion of type, cannot permit this operation, so the types that are passed and inspected at run time must be altered in the same way as all other types. This alteration disrupts the algorithmic structure of the program to no good purpose, and it also presents some severe practical problems:

- After the compiler transforms types, they usually become larger, often substantially. Passing and analyzing the altered types, instead of the original, leads to unnecessary inefficiency.
- The transformations are usually not surjective. Consequently, typecases that had been exhaustive before transformation can become inexhaustive, leaving the compiler to insert additional clauses to fill out every typecase. At best such clauses are wasteful; at worst they may be impossible to write in a type-safe manner.
- Sometimes the transformations are not even injective, making it impossible to appropriately transform typecase expressions in a meaning-preserving manner.

To solve these problems, we would like a type system that allows two distinct notions of type to coexist: the current types and the types used in some earlier stage of compilation. To clarify what we have in mind, we begin with a simple example. Consider the code fragment:

```
Λα:Type. λx:α.
  typecase α of
    int    => ... (* x has type int *) ...
    β × γ  => ... (* x has type β × γ *) ...
    β → γ  => ... (* x has type β → γ *) ...
```

Now suppose the compiler performs closure conversion [19, 24], thereby transforming function types $\tau_1 \to \tau_2$ into $\exists\delta.((\delta \times \tau_1) \to \tau_2) \times \delta$. In existing languages, this code must become the code in Figure 1.

Instead, we would like α to be a "high-level" type, but upon finding it to be $\beta \to \gamma$ we want to be able to conclude that x has the closure-converted type. Naively, then, we would like the language to supply two different kinds of types, Type (current types) and MLType (types before closure conversion), and a function interp : MLType $\to$ Type to translate between them. With these operations, we could transform the code fragment to something like Figure 2.

This naive language solves the hardwiring problem discussed above, but replaces it with another one. In this language the source's type system is hardwired (as MLType) and the type translation from the source is also hardwired (as interp). Thus, the language is defective as a general-purpose intermediate language; it specifies both the source language and the compilation strategy, and it ought to specify neither.

1.1 Our Solution

In this paper we introduce a new language, called LX, for expressing programs that analyze types. LX provides a very expressive type system in which one can *program* MLType and interp. In this manner we solve the hardwiring problem without having to specialize to a particular source language or compilation strategy. LX makes this solution possible by providing a rich programming language of type constructors. In this language, the kind MLType is definable using sum, product, and inductive kinds, and the operator interp is definable using primitive recursion.

Although LX was devised to support type analysis, it contains no constructs for analyzing types per se. This fact about LX reveals that intensional type analysis is simply a *programming idiom* that is possible in a language with sufficiently rich type constructors. The flexibility afforded by this language allows idioms going well beyond what has been previously possible in type-analyzing languages. In this paper we discuss three such applications:

- We present the first account of how to conduct intensional type analysis in the presence of polymorphic types and other types with binding structure.
- We show how to make "shallow" type analysis possible without passing entire types. This optimization is useful in applications where it is only necessary to determine the top-level structure of types, as in some garbage collectors.
- We illustrate an elegant way to express Haskell-style type classes [15] or ML equality types.

We also discuss another particularly important application of LX: As discussed in Crary, Weirich, and Morrisett [5] (hereafter, CWM), many aspects of compilation are greatly simplified by adopting a type-erasure semantics, but such a semantics seems problematic in the presence of type analysis. CWM reconciled type analysis with type-erasure semantics using explicit run-time terms to represent erasable type information in their language λ_R. In CWM those type representations were required to be primitive but we show that they are definable in LX.

The remainder of this paper is organized as follows: In Section 2 we discuss informally how to analyze types in LX. In Section 3 we formally define LX and state some important properties of it. In Section 4, we formally revisit the examples of Section 2, and also discuss polymorphic types, shallow type analysis and type classes. In Section 5 we show how to reconcile LX with a type-erasure semantics. Concluding discussion appears in Section 6. We assume some familiarity with the notions of type constructors and kinds.

2 Informal Presentation

We begin with a simple example to illustrate informally how type analysis is conducted in LX. Suppose we wish to store arrays of pairs efficiently. In a naive implementation, each pair in the array must be boxed so that array entries are uniformly word-sized. This representation wastes a word for every array entry, or more if the pair components are pairs themselves. We may store such arrays more efficiently by transforming them from arrays of pairs to pairs of arrays. This latter representation costs only a few words for the entire array.[1]

We would like the compiler to employ this optimization automatically for all arrays of pairs, including polymorphic arrays that happen to be arrays of pairs. This application is precisely the purpose of intensional type analysis; using intensional type analysis, a polymorphic function can analyze its type argument and dispatch to different code depending on that argument. To make what we mean concrete, we will first implement this optimization in the style of a conventional type analysis language, and then translate it into LX.

[1] An ever better representation would be to use arrays of unboxed, flattened tuples. This also can be done straightforwardly using type analysis [12], but is a more complicated example.

To implement this optimization, we define a type operator `optarray` and a corresponding subscript function `optsub` operating on optimized arrays. The `optarray` operator recursively splits arrays of pairs into pairs of arrays and uses ordinary arrays at all other types. We assume the built-in function `sub` has type $\forall\alpha.\,\texttt{array}\,\alpha \to \texttt{int} \to \alpha$.[2]

$$
\begin{aligned}
\texttt{optarray}(\texttt{int}) &\stackrel{\text{def}}{=} \texttt{array}(\texttt{int})\\
\texttt{optarray}(\tau_1 \times \tau_2) &\stackrel{\text{def}}{=} (\texttt{optarray}\,\tau_1) \times\\
&\phantom{\stackrel{\text{def}}{=}} (\texttt{optarray}\,\tau_2)\\
\texttt{optarray}(\tau_1 \to \tau_2) &\stackrel{\text{def}}{=} \texttt{array}(\tau_1 \to \tau_2)\\
\texttt{optarray}(\texttt{array}\,\tau) &\stackrel{\text{def}}{=} \texttt{array}(\texttt{array}\,\tau)
\end{aligned}
$$

```
val rec optsub : ∀α. optarray α → int → α =
  Fn [α] =>
    fn a : optarray α => fn n : int =>
      typecase α of
        β × γ => (optsub[β] (#1 a) n,
                  optsub[γ] (#2 a) n)
        _     => sub[α] a n
```

In an LX version of this example, `optarray` and `optsub` will no longer operate on types, they will operate on *type constructors* that encode types. In particular, we inductively define a kind MLType whose members specify the abstract syntax of a type. In this section we use an informal notation borrowed from ML datatypes; we will show how this example is formalized in the next section.

```
kind MLType = Int
            | Prod of MLType * MLType
            | Arrow of MLType * MLType
            | Array of MLType
```

Members of MLType have no built-in interpretation as types; they are merely data that may be computed with at the level of type constructors. The first thing to do then is to define their meaning by a function mapping MLType to Type:

$$
\begin{aligned}
\mathsf{interp}(\texttt{Int}) &\stackrel{\text{def}}{=} \texttt{int}\\
\mathsf{interp}(\texttt{Prod}(c_1, c_2)) &\stackrel{\text{def}}{=} \mathsf{interp}(c_1) \times \mathsf{interp}(c_2)\\
\mathsf{interp}(\texttt{Arrow}(c_1, c_2)) &\stackrel{\text{def}}{=} \mathsf{interp}(c_1) \to \mathsf{interp}(c_2)\\
\mathsf{interp}(\texttt{Array}(c)) &\stackrel{\text{def}}{=} \texttt{array}(\mathsf{interp}(c))
\end{aligned}
$$

Note that the function interp is primitive recursive. In order to ensure that computation with type constructors always terminates, arbitrary recursive functions are not permitted in LX, only primitive recursive ones.

Now that we have defined type encodings and their interpretations as actual types, we can proceed with the example as before. The new operator `OptArray` has kind MLType → MLType and is defined primitive recursively.

[2] While most of our examples resemble the syntax of ML, we use prefix notation for constructor application.

$$
\begin{array}{llll}
(\textit{kinds}) & k & ::= \mathsf{Type} \mid 1 \mid k_1 \to k_2 \mid k_1 \times k_2 \mid k_1 + k_2 \mid j \mid \mu j.k & \\
(\textit{constructors}) & c, \tau & ::= * \mid \alpha \mid \lambda\alpha{:}k.c \mid c_1 c_2 & \text{unit, variables and functions} \\
 & & \quad \mid \langle c_1, c_2 \rangle \mid \mathtt{prj}_1\, c \mid \mathtt{prj}_2\, c & \text{products} \\
 & & \quad \mid \mathtt{inj}_1^{k_1+k_2}\, c \mid \mathtt{inj}_2^{k_1+k_2}\, c \mid \mathtt{case}(c, \alpha_1.c_1, \alpha_2.c_2) & \text{sums} \\
 & & \quad \mid \mathtt{fold}_{\mu j.k}\, c \mid \mathtt{pr}(j, \alpha{:}k, \beta{:}j \to k'.c) & \text{primitive recursion} \\
 & & \quad \mid \mathtt{int} \mid \tau_1 \to \tau_2 \mid \tau_1 \times \tau_2 \mid \tau_1 + \tau_2 \mid \forall\alpha{:}k.\tau \mid \exists\alpha{:}k.\tau & \text{types} \\
 & & \quad \mid \mathtt{unit} \mid \mathtt{void} \mid \mathtt{rec}_k(c_1, c_2) & \text{types}
\end{array}
$$

Figure 3: LX Kinds and Constructors

The corresponding subscript function, `optsub`, now analyzes members of MLType rather than actual types.

$$
\begin{array}{rcl}
\mathtt{OptArray}(\mathtt{Int}) & \stackrel{\text{def}}{=} & \mathtt{Array}(\mathtt{Int}) \\
\mathtt{OptArray}(\mathtt{Prod}(c_1, c_2)) & \stackrel{\text{def}}{=} & \mathtt{Prod}(\mathtt{OptArray}(c_1), \\
 & & \qquad \mathtt{OptArray}(c_2)) \\
\mathtt{OptArray}(\mathtt{Arrow}(c_1, c_2)) & \stackrel{\text{def}}{=} & \mathtt{Array}(\mathtt{Arrow}(c_1, c_2)) \\
\mathtt{OptArray}(\mathtt{Array}(c)) & \stackrel{\text{def}}{=} & \mathtt{Array}(\mathtt{Array}(c))
\end{array}
$$

```
val rec optsub :
  ∀α:MLType. interp(OptArray(α))
              → int → interp α =
  Fn [α : MLType] =>
    fn a : interp(OptArray(α)) => fn n : int =>
      case α of
        Prod(β, γ) => (optsub[β] (#1 a) n,
                       optsub[γ] (#2 a) n)
        _          => sub[interp α] a n
```

Translating this example into LX has certainly made it more verbose, but it also makes it robust under further compilation. Suppose the compiler performs closure conversion, thereby transforming function types $\tau_1 \to \tau_2$ into $\exists\delta.((\delta \times \tau_1) \to \tau_2) \times \delta$. All that needs happen is a change to the appropriate clause of the interp function,

$$
\begin{array}{c}
\mathsf{interp}'(\mathtt{Arrow}(c_1, c_2)) \\
\stackrel{\text{def}}{=} \\
\exists\delta.((\delta \times \mathsf{interp}'(c_1)) \to \mathsf{interp}'(c_2)) \times \delta
\end{array}
$$

but no changes to `OptArray` or `optsub` are required (other than the closure conversion itself, of course).

3 A Language for Flexible Type Analysis

In this section we discuss LX and its semantics. We present the constructor and term levels individually, concentrating discussion on the novel features of each. The syntax of LX (shown in Figures 3 and 4) is based on Girard's F_ω [10, 9] augmented mainly by a rich programming language at the constructor level, and constructor refinement operators at the term level. The full static and operational semantics of LX are given in Appendices A and B.

3.1 Kinds and Constructors

The constructor and kind levels, shown in Figure 3, contain both base constructors of kind Type (called types) for classifying terms, and a variety of programming constructs for computing types. In addition to the variables and lambda abstractions of F_ω, LX also includes a unit kind, products, sums, and the usual introduction and elimination constructs for those kinds.

We denote the simultaneous, capture-avoiding substitution of $E_1, \ldots, E_n$ for $X_1, \ldots, X_n$ in E by $E[E_1, \ldots, E_n / X_1, \ldots, X_n]$. As usual, we consider alpha-equivalent expressions to be identical. A few constructs ($\mathtt{inj}_i$, `fold`, `pr`, and `rec`) are labeled with kinds to assist in kind checking; we will omit such kinds when they are clear from context. When a constructor is intended to have kind Type, we often use the metavariable τ.

To support computing with abstract syntax trees, LX includes kind variables (j) and inductive kinds ($\mu j.k$). A prospective inductive kind $\mu j.k$ will be well-formed provided that j appears only positively within k. Inductive kinds are formed using the introductory operator $\mathtt{fold}_{\mu j.k}$, which coerces constructors from kind $k[\mu j.k/j]$ to kind $\mu j.k$. For example, consider the kind of natural numbers N, defined as $\mu j.(1+j)$. The constructor $(\mathtt{inj}_1^{1+\mathsf{N}} *)$ has kind $(1+j)[\mathsf{N}/j]$. Therefore $\mathtt{fold}_{\mathsf{N}}(\mathtt{inj}_1^{1+\mathsf{N}} *)$ has kind N.

Inductive kinds are eliminated using the primitive recursion operator `pr`. Intuitively, $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)$ may be thought of as a recursive function with domain $\mu j.k$ in which α stands for the argument unfolded and φ recursively stands for the full function. However, in order to ensure that constructor expressions always terminate, we restrict `pr` to define only primitive recursive functions. Informally speaking, a function is primitive recursive if it can only call itself recursively on a subcomponent of its argument. Following Mendler [17], we ensure this using abstract kind variables. Since α stands for the argument unfolded, we could consider it to have the kind $k[\mu j.k/j]$, but instead of substituting for j in k, we hold j abstract. Then the recursive variable φ is given kind $j \to k'$ (instead of $j[\mu j.k/j] \to k'$) thereby ensuring that φ is called only on a subcomponent of α.

The kind k' in $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)$ is permitted to con-

(*terms*) e	$::= i \mid * \mid x \mid \lambda x{:}\tau.e \mid e_1 e_2$	ints, unit, variables, abstractions
	$\mid \langle e_1, e_2\rangle \mid \mathtt{prj}_1\, e \mid \mathtt{prj}_2\, e$	products
	$\mid \mathtt{inj}_1^{\tau_1+\tau_2}\, e \mid \mathtt{inj}_2^{\tau_1+\tau_2}\, e \mid \mathtt{case}(e, x_1.e_1, x_2.e_2)$	sums
	$\mid \Lambda\alpha{:}k.v \mid e[c] \mid \mathtt{fix}\, f{:}\tau.e$	constructor abstractions and recursion
	$\mid \mathtt{pack}\, e\, \mathtt{as}\, \exists\alpha{:}k.\tau\ \mathtt{hiding}\, c \mid \mathtt{unpack}\, \langle\alpha, x\rangle = e_1\, \mathtt{in}\, e_2$	existential packages
	$\mid \mathtt{fold}_{\mathtt{rec}_k(c,c')}\, e \mid \mathtt{unfold}\ e$	parameterized recursive types
	$\mid \mathtt{let}_\tau\ \langle\beta,\gamma\rangle = c\ \mathtt{in}\, e \mid \mathtt{let}_\tau\ (\mathtt{fold}\,\beta) = c\ \mathtt{in}\, e$	constructor refinement operations
	$\mid \mathtt{ccase}_\tau(c, \alpha_1.e_1, \alpha_2.e_2)$	

Figure 4: Terms in LX

tain (positive) free occurrences of j. In that case, the function's result kind employs the substitution for j that was internally eschewed. Hence, the result kind of the above constructor is $k'[\mu j.k/j]$. This is useful so that some part of the argument may be passed through without φ operating on it. As a particularly useful application, we can define the constructor $\mathtt{unfold}_{\mu j.k}$ with kind $\mu j.k \to k[\mu j.k/j]$ to be $\mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k.\alpha)$.

Given a constructor n with kind N, we can use primitive recursion to construct the type of $(n+1)$-tuples of integers (using an informal, expanded notation for `case`):

$$\begin{array}{lll}\mathsf{ntuple} & \stackrel{\text{def}}{=} & \mathtt{pr}(j, \alpha{:}1{+}j, \varphi{:}j \to \mathsf{Type}.\\ & & \quad \mathtt{case}\ \alpha\ \mathtt{of}\\ & & \qquad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{int}\\ & & \qquad \mathtt{inj}_2\,\gamma \Rightarrow \varphi(\gamma)\times\mathtt{int})\end{array}$$

Suppose we apply ntuple to $\overline{1}$ (that is, the encoding of the natural number 1, $\mathtt{fold}(\mathtt{inj}_2(\mathtt{fold}(\mathtt{inj}_1 *)))$). By unrolling the $\mathtt{pr}$ expression, we may show :

$$\begin{array}{l}(\mathtt{pr}(j, \alpha{:}1{+}j, \varphi{:}j \to \mathsf{Type}.\\ \quad \mathtt{case}\ \alpha\ \mathtt{of}\\ \qquad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{int}\\ \qquad \mathtt{inj}_2\,\gamma \Rightarrow \varphi(\gamma)\times\mathtt{int}))\,\overline{1}\\[1em] = \mathtt{case}\ (\mathtt{inj}_2(\mathtt{fold}(\mathtt{inj}_1 *)))\ \mathtt{of}\\ \qquad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{int}\\ \qquad \mathtt{inj}_2\,\gamma \Rightarrow \mathsf{ntuple}(\gamma)\times\mathtt{int}\\[1em] = (\mathsf{ntuple}(\mathtt{fold}(\mathtt{inj}_1 *)))\times\mathtt{int}\\[1em] = (\mathtt{case}\ (\mathtt{inj}_1 *)\ \mathtt{of}\\ \qquad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{int}\\ \qquad \mathtt{inj}_2\,\gamma \Rightarrow \mathsf{ntuple}(\gamma)\times\mathtt{int})\times\mathtt{int}\\[1em] = \mathtt{int}\times\mathtt{int}\end{array}$$

The unrolling process is formalized by the following constructor equivalence rule (the relevant judgment forms are summarized in Figure 5):

$$\frac{\begin{array}{cc}\Delta \vdash c' : k[\mu j.k/j] & \Delta, j \vdash k'\ \mathsf{kind}\\ \Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' & \Delta \vdash \mu j.k\ \mathsf{kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)(\mathtt{fold}_{\mu j.k}\, c') =\\ \quad c[\mu j.k, c', \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)/j, \alpha, \varphi]\\ \qquad\qquad : k'[\mu j.k/j]\end{array}}$$
$$(j \text{ only positive in } k' \text{ and } j, \alpha, \varphi \notin \Delta)$$

Notation 3.1 If k_1 is of the form $\mu j.k$, then we write $k_1[k_2]$ to mean $k[k_2/j]$.

3.2 Terms

The syntax of LX terms is given in Figure 4. Most LX terms are standard, including the usual introduction and elimination forms for functions, products, sums, unit, and universal and existential types. Constructor abstractions are limited by a value restriction, in anticipation of the type erasure interpretation in Section 5. The value forms of LX are given in Appendix B. Recursive functions are expressible using `fix` terms, the bodies of which are syntactically restricted to be functions (possibly polymorphic) by their typing rule (Appendix A). As at the constructor level, some constructs are labeled with types to assist in type checking; we omit these when clear from context.

Parameterized recursive types are written $\mathtt{rec}_k(c_1, c_2)$, where k is the parameter kind and c_1 is a type constructor with kind $(k \to \mathsf{Type}) \to (k \to \mathsf{Type})$. Intuitively, c_1 recursively defines a type constructor with kind $k \to \mathsf{Type}$, which is then instantiated with the parameter c_2 (having kind k). Thus, members of $\mathtt{rec}_k(c_1, c_2)$ unfold into the type $c_1(\lambda\alpha{:}k.\,\mathtt{rec}_k(c_1, \alpha))c_2$, and fold the opposite way. The special case of non-parameterized recursive types are defined as $\mathtt{rec}(\alpha.\tau) = \mathtt{rec}_1(\lambda\varphi{:}1 \to \mathsf{Type}.\,\lambda\beta{:}1.\,\tau[\varphi(*)/\alpha], *)$. Unlike inductive kinds, no positivity condition is imposed on recursive types.

Refinement The novel features of the LX term language are the three refinement operations. To perform constructor analysis at run time, we require a mechanism for branching on sum kinds at the term level. This branching is done using the `ccase` construct. If c normalizes to $\mathtt{inj}_1(c')$, then the term $\mathtt{ccase}(c, \alpha_1.e_1, \alpha_2.e_2)$ evaluates to $e_1[c'/\alpha_1]$, and similarly if it normalizes to $\mathtt{inj}_2(c')$.

However, we require more than a term that evaluates in the desired manner. After branching, we have learned something about the constructor in question, and this information may result in additional knowledge about the types of our data. We wish the type system to be able to exploit that knowledge. Consequently, the typing rule for `ccase`, when the constructor in question is some variable α, substitutes for α to propagate the new information:

$$\frac{\begin{array}{l}\Delta, \beta{:}k_1, \Delta'; \Gamma[\mathtt{inj}_1\,\beta/\alpha] \vdash \\ \qquad\qquad e_1[\mathtt{inj}_1\,\beta/\alpha] : \tau[\mathtt{inj}_1\,\beta/\alpha] \\ \Delta, \beta{:}k_2, \Delta'; \Gamma[\mathtt{inj}_2\,\beta/\alpha] \vdash \\ \qquad\qquad e_2[\mathtt{inj}_2\,\beta/\alpha] : \tau[\mathtt{inj}_2\,\beta/\alpha] \\ \quad \Delta, \alpha{:}k_1 + k_2, \Delta' \vdash c = \alpha : k_1 + k_2\end{array}}{\Delta, \alpha{:}k_1 + k_2, \Delta'; \Gamma \vdash \mathtt{ccase}_\tau(c, \beta.e_1, \beta.e_2) : \tau} \ (\beta \notin \Delta)$$

Within the branches, types that depend upon α can be reduced using the new information. For example, if x has type $\mathtt{case}(\alpha, \beta.\mathtt{int}, \beta.\mathtt{bool})$, its type can be reduced in either branch, allowing its use as an integer in one branch and as a boolean in the other.

In order for LX to enjoy the subject reduction property, we also require two *trivialization* rules [6] for `ccase`, for use when the argument to `ccase` is a sum introduction:

$$\frac{\Delta \vdash c = \mathtt{inj}_1\,c' : k_1 + k_2 \qquad \Delta; \Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta; \Gamma \vdash \mathtt{ccase}_\tau(c, \alpha.e_1, \alpha.e_2) : \tau}$$

$$\frac{\Delta \vdash c = \mathtt{inj}_2\,c' : k_1 + k_2 \qquad \Delta; \Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta; \Gamma \vdash \mathtt{ccase}_\tau(c, \alpha.e_1, \alpha.e_2) : \tau}$$

Path refinement There may also be useful refinement to perform when the constructor to be branched on is not a variable. For example, suppose α has kind $(1{+}1) \times \mathsf{Type}$ and x has type $\mathtt{case}(\mathtt{prj}_1\,\alpha, \beta.\mathtt{int}, \beta.\mathtt{bool})$. When branching on $\mathtt{prj}_1\,\alpha$, we should again be able to consider x an integer or boolean, but the ordinary `ccase` rule above no longer applies since $\mathtt{prj}_1\,\alpha$ is not a variable. This is solved using the product refinement operation, $\mathtt{let}_\tau\,\langle\beta, \gamma\rangle = \alpha\,\mathtt{in}\,e$. Like `ccase`, the product refinement operation substitutes everywhere for α:

$$\frac{\begin{array}{c}\Delta, \beta{:}k_1, \gamma{:}k_2, \Delta'; \Gamma[\langle\beta, \gamma\rangle/\alpha] \vdash e[\langle\beta, \gamma\rangle/\alpha] : \tau[\langle\beta, \gamma\rangle/\alpha] \\ \Delta, \alpha{:}k_1 \times k_2, \Delta' \vdash c = \alpha : k_1 \times k_2\end{array}}{\Delta, \alpha{:}k_1 \times k_2, \Delta'; \Gamma \vdash \mathtt{let}_\tau\langle\beta, \gamma\rangle = c\,\mathtt{in}\,e : \tau} \ (\beta, \gamma \notin \Delta)$$

A similar refinement operation exists for inductive types, and each operation also has a trivialization and a non-refining rule similar to those of `ccase`.

We may use these refinement operations to turn paths into variables and thereby take advantage of `ccase`. For example, suppose α has kind $\mathsf{N} \times \mathsf{N}$ and we wish to branch on $\mathtt{unfold}\ (\mathtt{prj}_1\,\alpha)$. We do it using product and inductive kind refinement in turn:

$$\begin{array}{l}\mathtt{let}\ \langle\beta_1, \beta_2\rangle = \alpha\ \mathtt{in} \\ \mathtt{let}\ (\mathtt{fold}\,\gamma) = \beta_1\ \mathtt{in} \\ \qquad \mathtt{ccase}(\gamma, \delta.e_1, \delta.e_2)\end{array}$$

Non-path refinement Since there is no refinement operation for functions, sometimes a constructor cannot be reduced to a path. Nevertheless, it is still possible to gain some of the benefits of refinement, using a device due to Harper and Morrisett [12]. Suppose φ has kind $\mathsf{N} \to (1{+}1)$, x has type $\mathtt{case}(\varphi(\bar{1}), \beta.\mathtt{int}, \beta.\mathtt{bool})$, and we wish to branch on $\varphi(\bar{1})$ to learn the type of x. First we use a constructor abstraction to assign a variable α to $\varphi(\bar{1})$, thereby enabling `ccase`, and then we use an ordinary abstraction to rebind x with type $\mathtt{case}(\alpha, \beta.\mathtt{int}, \beta.\mathtt{bool})$:

$$\begin{array}{l}(\Lambda\alpha{:}1{+}1.\ \lambda x{:}\,\mathtt{case}(\alpha, \beta.\mathtt{int}, \beta.\mathtt{bool}). \\ \quad \mathtt{ccase}_\tau(\alpha, \beta.e_1, \beta.e_2))\ [\varphi(\bar{1})]\ x\end{array}$$

Within e_1, x will be an integer, and similarly within e_2. This device has all the expressive power of refinement, but is less efficient because of the need for extra beta-expansions. However, this is the best that can be done with unknown functions.

3.3 Properties of LX

Judgment	Meaning
$\Delta \vdash k$ kind	k is a well-formed kind
$\Delta \vdash c : k$	c is a valid constructor of kind k
$\Delta \vdash c_1 = c_2 : k$	c_1 and c_2 are equal constructors
$\Delta; \Gamma \vdash e : \tau$	e is a term of type τ

Contexts

$$\begin{array}{lcl}\Delta & ::= & \epsilon \mid \Delta, j \mid \Delta, \alpha{:}k \\ \Gamma & ::= & \epsilon \mid \Gamma, x{:}\tau\end{array}$$

Figure 5: Judgments of LX

The judgments of the static semantics of LX appear in Figure 5. The important properties to show are decidable type checking and type safety. Due to space considerations, we do not present proofs of these properties here; details appear in the companion technical report [4]. For typechecking, the challenging part is deciding equality of type constructors. We do this using a normalize and compare method employing a reduction relation extracted from the equality rules in the obvious manner.

Lemma 3.2 *Reduction of well-formed constructors is strongly normalizing, confluent, preserves kinds, and is respected by equality.*

Strong normalization is proven using Mendler's variation on Girard's method [17]. Given Lemma 3.2 it is easy to show the normalize and compare algorithm to be terminating, sound and complete, and decidability of type checking follows in a straightforward manner.

Theorem 3.3 (Decidability) *It is decidable whether or not* $\Delta; \Gamma \vdash e : \tau$ *is derivable in LX.*

We say that a term is *stuck* if it is not a value and if no rule of the operational semantics applies to it. Type safety requires that no well-typed term can become stuck:

Theorem 3.4 (Type Safety) *If $\emptyset \vdash e : \tau$ and $e \mapsto^* e'$ then e' is not stuck.*

This is shown using the usual subject reduction and progress lemmas.

4 Programming Type Analysis

In this section, we discuss how to implement type analysis in general and as a specific example we formalize the example from Section 2. We then show how to extend this formulation through simple modifications to implement applications of type analysis that were previously inexpressible.

The basic idea of the type analysis programming idiom is to use elements of the constructor language to represent types, and to define an interpretation function such that at any point the type it represents may be extracted. Instead of destructing types through an additional language construct, as in Harper and Morrisett [12] or CWM, the representations are examined with the built-in features of LX.

Recall the kind MLType and its interpretation function from Section 2:

```
kind MLType = Int
            | Prod of MLType * MLType
            | Arrow of MLType * MLType
            | Array of MLType
```

$$\begin{array}{rcl}
\mathsf{interp}(\mathtt{Int}) & \stackrel{\text{def}}{=} & \mathtt{int} \\
\mathsf{interp}(\mathtt{Prod}(c_1, c_2)) & \stackrel{\text{def}}{=} & \mathsf{interp}(c_1) \times \mathsf{interp}(c_2) \\
\mathsf{interp}(\mathtt{Arrow}(c_1, c_2)) & \stackrel{\text{def}}{=} & \mathsf{interp}(c_1) \to \mathsf{interp}(c_2) \\
\mathsf{interp}(\mathtt{Array}(c)) & \stackrel{\text{def}}{=} & \mathtt{array}(\mathsf{interp}(c))
\end{array}$$

If we add an array type constructor to LX for this example, we can formalize these definitions in LX by melding the datatype definition of MLType into a recursive sum of products:

in the following manner:

$$\begin{array}{l}
\mathsf{MLType} \stackrel{\text{def}}{=} \mu j.(1 + (j \times j) + (j \times j) + j) \\
\mathsf{interp} \stackrel{\text{def}}{=} \\
\quad \mathtt{pr}(j, \alpha{:}\mathsf{MLType}[j], \varphi{:}j \to \mathsf{Type}. \\
\quad\quad \mathsf{case}\,\alpha\,\mathsf{of} \\
\quad\quad\quad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{int} \\
\quad\quad\quad \mathtt{inj}_2\,\beta \Rightarrow \\
\quad\quad\quad\quad (\mathsf{case}\,\beta\,\mathsf{of} \\
\quad\quad\quad\quad\quad \mathtt{inj}_1\,\beta \Rightarrow \varphi(\mathtt{prj}_1\,\beta) \times \varphi(\mathtt{prj}_2\,\beta) \\
\quad\quad\quad\quad\quad \mathtt{inj}_2\,\beta \Rightarrow \\
\quad\quad\quad\quad\quad\quad (\mathsf{case}\,\beta\,\mathsf{of} \\
\quad\quad\quad\quad\quad\quad\quad \mathtt{inj}_1\,\beta \Rightarrow \varphi(\mathtt{prj}_1\,\beta) \to \varphi(\mathtt{prj}_2\,\beta) \\
\quad\quad\quad\quad\quad\quad\quad \mathtt{inj}_2\,\beta \Rightarrow \mathtt{array}(\varphi(\beta)))))
\end{array}$$

Now recall the function optsub from Section 2. To formalize optsub in LX, we need ccase and inductive kind refinement:

$$\begin{array}{l}
\mathtt{optsub} \stackrel{\text{def}}{=} \\
\mathtt{fix\,optsub} : (\forall\alpha{:}\mathsf{MLType}.\,\mathsf{interp}(\mathtt{OptArray}(\alpha)) \to \\
\qquad\qquad\qquad\qquad\qquad \mathtt{int} \to \mathsf{interp}\,\alpha). \\
\quad \Lambda\alpha{:}\mathsf{MLType}.\,\lambda\mathtt{a}{:}\,\mathsf{interp}(\mathtt{OptArray}(\alpha)).\,\lambda\mathtt{n}{:}\mathtt{int}. \\
\quad\quad \mathtt{let}_{(\mathsf{interp}\,\alpha)}\,(\mathtt{fold}\,\alpha') = \alpha\;\mathtt{in} \\
\quad\quad\quad \mathtt{ccase}_{(\mathsf{interp}\,\alpha)}\;\alpha'\;\mathtt{of} \\
\quad\quad\quad\quad \mathtt{inj}_1\,\beta \Rightarrow \mathtt{sub}[\mathsf{interp}\,\alpha]\;\mathtt{a}\;\mathtt{n} \\
\quad\quad\quad\quad \mathtt{inj}_2\,\beta \Rightarrow \\
\quad\quad\quad\quad\quad (\mathtt{ccase}_{(\mathsf{interp}\,\alpha)}\;\beta\,\mathtt{of} \\
\quad\quad\quad\quad\quad\quad \mathtt{inj}_1\,\gamma \Rightarrow \langle \mathtt{optsub}[\mathtt{prj}_1\,\gamma]\;(\mathtt{prj}_1\,\mathtt{a})\;\mathtt{n}, \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathtt{optsub}[\mathtt{prj}_2\,\gamma]\;(\mathtt{prj}_2\,\mathtt{a})\;\mathtt{n}\rangle \\
\quad\quad\quad\quad\quad\quad \mathtt{inj}_2\,\gamma \Rightarrow \ldots)
\end{array}$$

Let us verify that optsub is well-typed using the typing rules from the previous section. The interesting branch is the one dealing with products (beginning with "$\mathtt{inj}_1\,\gamma \Rightarrow \ldots$"). The let operation creates a new variable α' with kind MLType[MLType] and substitutes $\mathtt{fold}(\alpha')$ everywhere that α appears. In the product branch, after two uses of ccase, γ has kind $\mathsf{MLType} \times \mathsf{MLType}$ and $\mathtt{inj}_2(\mathtt{inj}_1(\gamma))$ is substituted for α'.

The required result type is $\mathsf{interp}\,\alpha$, which (after substitution) has become $\mathsf{interp}(\mathtt{fold}(\mathtt{inj}_2(\mathtt{inj}_1(\gamma))))$, which in turn is equal to $\mathsf{interp}(\mathtt{prj}_1\,\gamma) \times \mathsf{interp}(\mathtt{prj}_2\,\gamma)$. The type of a is $\mathsf{interp}(\mathtt{OptArray}(\alpha))$, which has become $\mathsf{interp}(\mathtt{OptArray}(\mathtt{fold}(\mathtt{inj}_2(\mathtt{inj}_1(\gamma)))))$, which in turn is equal to $\mathsf{interp}(\mathtt{OptArray}(\mathtt{prj}_1\,\gamma)) \times \mathsf{interp}(\mathtt{OptArray}(\mathtt{prj}_2\,\gamma))$. Thus $\mathtt{prj}_1\,\mathtt{a}$ and $\mathtt{prj}_2\,\mathtt{a}$ have the appropriate type and the branch typechecks.

Clearly the official LX syntax is quite verbose, so we will use the datatype-style notation in what follows.

4.1 Types with Binding Structure

Previous accounts of intentional type analysis have been unable to deal with types with binding structure, such as universal, existential or recursive types. In LX it is easy to deal with binding structure, simply by appropriate programming.

For example, we can encode the polymorphic lambda calculus using de Bruijn indices as follows:

```
kind FType = Var of N
           | Arrow of FType * FType
           | Forall of FType
```

To interpret an FType we also need to provide an environment ρ that maps type variables (natural numbers) to types, thus interp will have kind $\mathsf{FType} \to (\mathsf{N} \to \mathsf{Type}) \to \mathsf{Type}$. In the variable case, we just look it up in the environment, and in the $\forall$ branch, we interpret the body with an appropriately extended environment.

$$\begin{array}{rcl}
\mathsf{interp}(\mathtt{Var}(c)) & \stackrel{\text{def}}{=} & \lambda\rho{:}\mathsf{N}\to\mathsf{Type}.\,\rho(c) \\
\mathsf{interp}(\mathtt{Arrow}(c_1,c_2)) & \stackrel{\text{def}}{=} & \lambda\rho{:}\mathsf{N}\to\mathsf{Type}. \\
& & \quad \mathsf{interp}(c_1)(\rho)\to \\
& & \quad \mathsf{interp}(c_2)(\rho) \\
\mathsf{interp}(\mathtt{Forall}(c)) & \stackrel{\text{def}}{=} & \lambda\rho{:}\mathsf{N}\to\mathsf{Type}.\,\forall\alpha{:}\mathsf{Type}. \\
& & \quad \mathsf{interp}(c) \\
& & \quad (\lambda\beta{:}\mathsf{N}. \\
& & \qquad \mathtt{case\,unfold}\ \beta\ \mathtt{of} \\
& & \qquad\quad \mathtt{inj}_1\,\gamma \Rightarrow \alpha \\
& & \qquad\quad \mathtt{inj}_2\,\gamma \Rightarrow \rho(\gamma))
\end{array}$$

Type analysis of this language at the term level can be defined in a similar manner to the previous example.

It is important to note that this technique is limited to *parametrically* polymorphic functions, and cannot account for functions that perform intensional type analysis. It seems possible that through more complicated LX programming one might account for *some* functions that analyze types, but recent results [7] suggest that complete bootstrapping is probably impossible.

4.2 Shallow Representations

Some applications of type analysis are "shallow," and rely on the outermost structure of the type only, and not on its subcomponents. For example, a tag-free garbage collector needs to know if a given location is a pointer to code, but may not need the types of the arguments to that code [29, 3].

However, even though at run time only part of the type information might be used, the interpretation function interp must be able to reconstruct the entire type. We can implement this by including the type itself in the representation. The following definition, SType, describes representations that do not support analysis of function domains or codomains:

```
kind SType = Int
           | Prod of SType * SType
           | Arrow of Type * Type
```

Because the types appear literally in the constructors, the interpretation function does not need to recur in the third branch.

$$\begin{array}{rcl}
\mathsf{interp}(\mathtt{Int}) & \stackrel{\text{def}}{=} & \mathtt{int} \\
\mathsf{interp}(\mathtt{Prod}(c_1,c_2)) & \stackrel{\text{def}}{=} & \mathsf{interp}(c_1)\times\mathsf{interp}(c_2) \\
\mathsf{interp}(\mathtt{Arrow}(\tau_1,\tau_2)) & \stackrel{\text{def}}{=} & \tau_1\to\tau_2
\end{array}$$

In the formulation of the type erasable version of LX in Section 5, we will see that the unused portion of the type can indeed be erased and so will not be passed at runtime.

4.3 Type Classes

Some applications of type analysis may wish to limit analysis only to a subset of the types of the language. A canonical example of this sort of application is polymorphic equality in ML, an operation that is defined on only those data objects that admit equality, such as integers, booleans, and lists, but not functions. Also, the language Haskell [15] provides a general mechanism for defining classes of types with associated operations on them.

Previous type analyzing languages have implemented non-total dynamic type dispatch through the use of a "characteristic function" over the domain. This function is defined to be the identity at types that are allowed, and `void` elsewhere. For example, Harper and Morrisett define the class of types that admit equality using `Typerec` as (assuming the addition of the type `bool`):

$$\begin{array}{rcl}
\mathsf{Eq}(\mathtt{int}) & \stackrel{\text{def}}{=} & \mathtt{int} \\
\mathsf{Eq}(\mathtt{bool}) & \stackrel{\text{def}}{=} & \mathtt{bool} \\
\mathsf{Eq}(c_1\times c_2) & \stackrel{\text{def}}{=} & \mathsf{Eq}(c_1)\times\mathsf{Eq}(c_2) \\
\mathsf{Eq}(c_1\to c_2) & \stackrel{\text{def}}{=} & \mathtt{void}
\end{array}$$

With this predicate, they define a polymorphic equality function `eq` with type $\forall\alpha{:}\mathsf{Type}.\,\mathsf{Eq}\,\alpha \to \mathsf{Eq}\,\alpha \to \mathtt{bool}$ recursively dispatching to primitive equality functions and providing a trivial function with type $\mathtt{void}\to\mathtt{void}$ at illegal types. However, this encoding is not entirely satisfactory because $\mathtt{eq}[c_1 \to c_2]$ can be a well-typed expression. The function resulting from evaluation of this expression can only be applied to values of type `void`, so this function cannot be used, but we would prefer the type error to be generated at the point of instantiation, not application.

LX, on the other hand, can define the kind EqType as

```
kind EqType = Int
            | Bool
            | Prod of EqType * EqType
```

representing integers, booleans, and products of EqTypes, but not including function types. If `eq` has type $\forall\alpha{:}\,\mathsf{EqType}\,.\,(\mathsf{interp}\,\alpha)\to(\mathsf{interp}\,\alpha)\to\mathtt{bool}$, where interp is defined similarly to before, it is simply impossible to instantiate it illegally at a function type.

5 Type Erasure

The most important contribution of CWM is its reconciliation of type analysis with type-erasure semantics, through the use of primitive terms that express the representations of types at run time. This mechanism allows a semantics where types and type constructors may be erased, as their representations remain to be examined. Accounting for type erasure is an important step in extending type analysis to low-level languages.

What prevents type erasure in LX as presented thus far is the `ccase` construct: evaluation of `ccase` depends on its argument constructor. However, sometimes it is possible to know at compile-time which branch the `ccase` will take from the types of the branches. For example, if a branch produces a value of type `void`, we can infer that it is never taken as there are no values of that type.

We can form a type-erasable version of LX by requiring this always to happen. In particular, we replace the `ccase` construct with `vcase` (virtual case), in which one branch is required to be dead code (and is so marked), but which is otherwise identical. Since the dead branch is marked syntactically, the operational semantics need not examine the constructor argument, in a sense that is made precise in Appendix C. The formation rule for `vcase` with a dead left branch is (the right case is similar):

$$\frac{\begin{array}{l}\Delta, \beta{:}k_1, \Delta'; \Gamma[\mathtt{inj}_1\,\beta/\alpha] \vdash \\ \qquad v[\mathtt{inj}_1\,\beta/\alpha] : \mathtt{void} \\ \Delta, \beta{:}k_2, \Delta'; \Gamma[\mathtt{inj}_2\,\beta/\alpha] \vdash \\ \qquad e[\mathtt{inj}_2\,\beta/\alpha] : \tau[\mathtt{inj}_2\,\beta/\alpha] \\ \Delta, \alpha{:}k_1 + k_2, \Delta' \vdash c = \alpha : k_1 + k_2\end{array}}{\Delta, \alpha{:}k_1 + k_2, \Delta'; \Gamma \vdash \mathtt{vcase}_\tau(c, \beta.\,\mathtt{dead}\, v, \beta.e) : \tau} \quad (\beta \notin \Delta)$$

We list the complete rules for `vcase` in Appendix A.5.

This restriction would seem to reduce the expressive power of the language, but as in CWM, we can use representation terms to capture the structure of the constructors being erased. However, unlike CWM, in LX these representation terms are programmable without adding any new mechanisms. For example, a unit constructor is represented by the unit term, and a pair of constructors is represented by a pair of terms, and so forth.

This idea is formalized in Figures 6 and 7. If c is a constructor with kind k, then $\ulcorner c \urcorner$ is its representation and that representation has type $R(c : k)$. Note that types have trivial representations so they cannot be analyzed, but this is no loss since types are not directly analyzable in full LX either.

The following proposition makes precise the notion that a constructor's representation does represent it, by stating that in an appropriate context, the translation of a constructor has the correct type:

Proposition 5.1 *Define R_{con} and R_{val} as:*

$$\begin{array}{rcl} R_{con}(\epsilon) & \stackrel{\text{def}}{=} & \epsilon \\ R_{val}(\epsilon) & \stackrel{\text{def}}{=} & \epsilon \\ R_{con}(\Delta, j) & \stackrel{\text{def}}{=} & R_{con}(\Delta), j, \varphi_j{:}j \to \mathsf{Type} \\ R_{val}(\Delta, j) & \stackrel{\text{def}}{=} & R_{val}(\Delta) \\ R_{con}(\Delta, \alpha{:}k) & \stackrel{\text{def}}{=} & R_{con}(\Delta), \alpha{:}k \\ R_{val}(\Delta, \alpha{:}k) & \stackrel{\text{def}}{=} & R_{val}(\Delta), x_\alpha{:}R(\alpha : k) \end{array}$$

If $\Delta \vdash c : k$ then $R_{con}(\Delta); R_{val}(\Delta) \vdash \ulcorner c \urcorner : R(c : k)$.

$$\begin{array}{rcl} R(c : 1) & \stackrel{\text{def}}{=} & \mathtt{unit} \\ R(c : k_1 \to k_2) & \stackrel{\text{def}}{=} & \forall \alpha{:}k_1. R(\alpha : k_1) \to R(c\alpha : k_2) \\ & & \text{(where } \alpha \text{ is fresh)} \\ R(c : k_1 \times k_2) & \stackrel{\text{def}}{=} & R(\mathtt{prj}_1\, c : k_1) \times R(\mathtt{prj}_2\, c : k_2) \\ R(c : k_1 + k_2) & \stackrel{\text{def}}{=} & \mathtt{case}(c, \alpha.R(\alpha : k_1), \alpha.\mathtt{void}) + \\ & & \mathtt{case}(c, \alpha.\mathtt{void}, \alpha.R(\alpha : k_2)) \\ R(c : j) & \stackrel{\text{def}}{=} & \varphi_j c \\ R(c : \mu j.k) & \stackrel{\text{def}}{=} & \mathtt{rec}_{\mu j.k}(\lambda \varphi_j{:}\mu j.k \to \mathsf{Type}. \\ & & \quad \lambda \alpha{:}\mu j.k.R(\mathtt{unfold}\ \alpha : k), c) \\ & & \text{(where } \alpha \text{ is fresh)} \\ R(c : \mathsf{Type}) & \stackrel{\text{def}}{=} & \mathtt{unit} \end{array}$$

Figure 6: Representation types

It remains to show that representation terms are sufficient for simulating `ccase` using `vcase`. Suppose c has kind $k_1 + k_2$. Then $\ulcorner c \urcorner$ has type $R(c : k_1 + k_2)$. Branching on $\ulcorner c \urcorner$ provides a value with type $\mathtt{case}(c, \beta.R(\beta{:}k_1), \beta.\mathtt{void})$ or with the converse type. A value with the given type determines that c must be a left injection, because the other choice provides an impossible value of type `void`. A value with the converse type similarly determines c to be a right injection. Either way, we can propagate this information into the type system using `vcase`. To make this intuition precise, observe that any well-typed term of the form $\mathtt{ccase}_\tau(c, \alpha.e_1, \alpha.e_2)$ can be replaced by the term

$$\begin{array}{l} \mathtt{case}\, \ulcorner c \urcorner\ \mathtt{of} \\ \quad \mathtt{inj}_1\, x \Rightarrow \mathtt{vcase}_\tau(c, \alpha.e_1, \alpha.\,\mathtt{dead}\, x) \\ \quad \mathtt{inj}_2\, x \Rightarrow \mathtt{vcase}_\tau(c, \alpha.\,\mathtt{dead}\, x, \alpha.e_2) \end{array}$$

provided that representations for every free variable of c are in scope, as required by Proposition 5.1.

This strategy can be used to encode the entire λ_R language of CWM into the erasable version of LX, demonstrating that LX has the full expressive power of previous type-analyzing languages. Space considerations prevent us from including the complete details of the encoding here; those details appear in the companion technical report [4].

6 Related Work and Conclusions

The properties and applications of languages with inductive types similar to the constructor level of LX have been well-studied by Mendler [18, 17], Werner [31], Howard [13, 14], and Gordon [11], among others. Most of those studies include coinductive and polymorphic types as well as inductive types. It appears as though extending LX with coinductive and polymorphic kinds would not be problematic. We have omitted such extensions at present in order to simplify the language and because it is not immediately clear how useful such extensions would be.

Duggan [8] proposes another typed framework for intensional type analysis that is similar in some ways to LX.

$$
\begin{array}{rcl}
\ulcorner * \urcorner & \stackrel{\text{def}}{=} & * \\
\ulcorner \alpha \urcorner & \stackrel{\text{def}}{=} & x_\alpha \\
\ulcorner \lambda\alpha{:}k.c \urcorner & \stackrel{\text{def}}{=} & \Lambda\alpha{:}k.\,\lambda x_\alpha{:}R(\alpha : k).\,\ulcorner c\urcorner \\
\ulcorner c_1 c_2 \urcorner & \stackrel{\text{def}}{=} & \ulcorner c_1\urcorner[c_2]\ulcorner c_2\urcorner \\
\ulcorner\langle c_1,c_2\rangle\urcorner & \stackrel{\text{def}}{=} & \langle\ulcorner c_1\urcorner,\ulcorner c_2\urcorner\rangle \\
\ulcorner \mathtt{prj}_i\,c\urcorner & \stackrel{\text{def}}{=} & \mathtt{prj}_i\ulcorner c\urcorner \\
\ulcorner\mathtt{inj}_i^{k_1+k_2}\,c\urcorner & \stackrel{\text{def}}{=} & \mathtt{inj}_i^{R(\mathtt{inj}_i\,c:k_1+k_2)}\ulcorner c\urcorner \\
\ulcorner\mathtt{case}(c,\alpha.c_1,\alpha.c_2)\urcorner & \stackrel{\text{def}}{=} & (\Lambda\beta{:}k_1+k_2.\,\lambda x{:}R(\beta{:}k_1+k_2). \\
 & & \quad \mathtt{case}\,x\,\mathtt{of} \\
 & & \qquad \mathtt{inj}_1\,x_\alpha \Rightarrow \mathtt{vcase}_{R(\mathtt{case}(\beta,\alpha.c_1,\alpha.c_2):k)}(\beta,\alpha.\ulcorner c_1\urcorner,\alpha.\,\mathtt{dead}\,x_\alpha) \\
 & & \qquad \mathtt{inj}_2\,x_\alpha \Rightarrow \mathtt{vcase}_{R(\mathtt{case}(\beta,\alpha.c_1,\alpha.c_2):k)}(\beta,\alpha.\,\mathtt{dead}\,x_\alpha,\alpha.\ulcorner c_2\urcorner))[c]\ulcorner c\urcorner \\
 & & (\text{where } \beta \text{ is fresh, } k_1+k_2 \text{ is the kind of } c\text{, and } k \text{ is the kind of } \mathtt{case}(c,\alpha.c_1,\alpha.c_2)) \\
\ulcorner\mathtt{fold}_{\mu j.k}\,c\urcorner & \stackrel{\text{def}}{=} & \mathtt{fold}_{R(c:\mu j.k)}\ulcorner c\urcorner \\
\ulcorner\mathtt{pr}(j,\alpha{:}k,\varphi{:}j\to k'.c)\urcorner & \stackrel{\text{def}}{=} & \mathtt{fix}\,x_\varphi : R(\mathtt{pr}(j,\alpha{:}k,\varphi{:}j\to k'.c) : \mu j.k \to k'[\mu j.k/j]). \\
 & & \quad \Lambda\beta{:}\mu j.k.\,\lambda x{:}R(\beta : \mu j.k). \\
 & & \qquad (\lambda x_\alpha{:}R(\mathtt{unfold}\,\beta : k[\mu j.k/j]). \\
 & & \qquad\quad \ulcorner c\urcorner[\mu j.k,(\lambda\gamma{:}\mu j.k.R(\gamma{:}\mu j.k)),\mathtt{unfold}\,\beta,\mathtt{pr}(j,\alpha{:}k,\varphi{:}j\to k'.c)/j,\varphi_j,\alpha,\varphi]) \\
 & & \qquad (\mathtt{unfold}\,x) \\
 & & (\text{where } \beta \text{ is fresh}) \\
\ulcorner\mathtt{int}\urcorner,\ulcorner\tau_1\to\tau_2\urcorner,\ldots & \stackrel{\text{def}}{=} & *
\end{array}
$$

Figure 7: Representation terms

Duggan's system passes types implicitly and primitively allows for the intensional analysis of types at the term level, but does not support intensional type analysis at the constructor level. It does add a facility for defining type classes (using union and recursive kinds) and allows type analysis to be restricted to members of such classes.

Morrisett *et al.* [24] developed typing mechanisms for low-level intermediate and target languages that allow type information to be preserved all the way to the end of compilation. It would be desirable, in a system based on those mechanisms, to exploit that type information using intensional type analysis. While CWM extended type analysis to the type-erasure semantics necessary for low-level typing mechanisms, remaining issues have prevented the use of the mechanisms of Morrisett *et al.* in type-analyzing compilers such as TIL/ML [20, 29] and FLINT [27, 28], and have made it as yet infeasible to use intensional type analysis in an end-to-end typed compiler.

The ambition of our work is to lay the foundation for an end-to-end typed compiler that supports intensional type analysis. LX provides a type-theoretic framework that supports the passing and analysis of type information at run time, but without native type analysis constructs. Because type analysis must be programmed within LX, much flexibility in the type system analyzed is afforded, resolving many of the issues hindering type analysis in later stages of typed compilation.

In pursuance of the aim of a type-analyzing end-to-end compiler, an important direction for future work is to extend the mechanisms of LX into lower-level typed assembly languages, and create a type-analyzing Typed Assembly Language. To evaluate this system in the framework of compilation, we plan to extend the Popcorn compiler and its target language TALx86 [22] to support type analysis.

References

[1] Shail Aditya, Christine Flood, and James Hicks. Garbage collection for strongly-typed languages using run-time type reconstruction. In *1994 ACM Conference on Lisp and Functional Programming*, pages 12–23, Orlando, June 1994.

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, Florida, January 1996.

[3] Perry Cheng, Robert Harper, and Peter Lee. Generational stack collection and profile-driven pretenuring. In *1998 SIGPLAN Conference on Programming Language Design and Implementation*, pages 162–173, 1998.

[4] Karl Crary and Stephanie Weirich. Flexible type analysis (extended version). Technical report, Cornell University, 1999.

[5] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. In *1998 ACM International Conference on Functional Programming*, pages 301–312, Baltimore, September 1998. Extended version published as Cornell University technical report TR98-1721.

[6] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type-erasure semantics. Technical Report TR98-1721, Cornell University, 1998.

[7] Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on*

Typed Lambda Calculi and Applications, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163, Nancy, France, April 1997. Springer-Verlag. Extended version published as CMU technical report CMU-CS-96-172.

[8] Dominic Duggan. A type-based semantics for user-defined marshalling in polymorphic languages. In *Second Workshop on Types in Compilation*, March 1998.

[9] Jean-Yves Girard. Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[10] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[11] Andrew D. Gordon. *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press, 1994.

[12] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[13] Brian T. Howard. *Fixed Points and Extensionality in Typed Functional Programming Languages.* PhD thesis, Stanford University, 1992. Published as Stanford Computer Science Department Technical Report STAN-CS-92-1455.

[14] Brian T. Howard. Inductive, coinductive, and pointed types. In *Proceedings of the 1996 ACM SIGPLAN International Conference on Functional Programming*, pages 102–109, 1996.

[15] Paul Hudak, Simon L. Peyton Jones, and Philip Wadler. Report on the programming language Haskell, version 1.2. *SIGPLAN Notices*, 27(5), May 1992.

[16] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[17] Nax Paul Mendler. Inductive types and type constraints in the second-order lambda calculus. *Annals of Pure and Applied Logic*, 51(1–2):159–172, 1991. Earlier version in LICS '88. (pp. 15,19,134,135).

[18] Paul Francis Mendler. *Inductive Definition in Type Theory.* PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.

[19] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[20] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[21] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[22] Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Fred Smith, Dave Walker, Stephanie Weirich, and Steve Zdancewic. TALx86: A realistic typed assembly language. Technical report, Cornell University, 1999. Submitted to the 1999 ACM SIGPLAN Workshop on Compiler Support for System Software.

[23] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics.* Cambridge University Press, 1997.

[24] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems*, 1999. To appear. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.

[25] Erik Ruf. Partitioning dataflow analyses using types. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 15–26, Paris, January 1997.

[26] Zhong Shao. Flexible representation analysis. In *1997 ACM International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[27] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[28] Zhong Shao, Christopher League, and Stefan Monnier. Implementing typed intermediate languages. In *1998 ACM International Conference on Functional Programming*, pages 313–323, Baltimore, Maryland, September 1998.

[29] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[30] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *1994 ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.

[31] Benjamin Werner. *Une Théorie des Constructiones Inductives.* PhD thesis, Université Paris VII, 1994.

A Static Semantics

A.1 Kind formation

$\boxed{\Delta \vdash k \ \mathsf{kind}}$

$$\frac{}{\Delta \vdash \mathsf{Type}\ \mathsf{kind}}$$

$$\frac{}{\Delta \vdash 1\ \mathsf{kind}}$$

$$\frac{}{\Delta \vdash j\ \mathsf{kind}}\ (j \in \Delta)$$

$$\frac{\Delta, j \vdash k\ \mathsf{kind}}{\Delta \vdash \mu j.k\ \mathsf{kind}}\ \begin{pmatrix} j \text{ only positive in } k \\ j \notin \Delta \end{pmatrix}$$

$$\frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 \to k_2\ \mathsf{kind}}$$

$$\frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 + k_2\ \mathsf{kind}}$$

$$\frac{\Delta \vdash k_1\ \mathsf{kind} \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash k_1 \times k_2\ \mathsf{kind}}$$

A.2 Constructor Formation

$\boxed{\Delta \vdash c : k}$

$$\frac{}{\Delta \vdash * : 1}$$

$$\frac{}{\Delta \vdash \alpha : \Delta(\alpha)}$$

$$\frac{\Delta, \alpha{:}k' \vdash c : k \qquad \Delta \vdash k' \ \mathsf{kind}}{\Delta \vdash \lambda\alpha{:}k'.c : k' \to k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c_1 : k' \to k \qquad \Delta \vdash c_2 : k'}{\Delta \vdash c_1 c_2 : k}$$

$$\frac{\Delta \vdash c_1 : k_1 \qquad \Delta \vdash c_2 : k_2}{\Delta \vdash \langle c_1, c_2 \rangle : k_1 \times k_2}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_1\, c : k_1}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_2\, c : k_2}$$

$$\frac{\Delta \vdash c : k_1 \qquad \Delta \vdash k_2 \ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_1^{k_1+k_2}\, c : k_1 + k_2}$$

$$\frac{\Delta \vdash c : k_2 \qquad \Delta \vdash k_1 \ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_2^{k_1+k_2}\, c : k_1 + k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c : k_1 + k_2 \\ \Delta, \alpha{:}k_1 \vdash c_1 : k \\ \Delta, \alpha{:}k_2 \vdash c_2 : k\end{array}}{\Delta \vdash \mathtt{case}(c, \alpha.c_1, \alpha.c_2) : k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k}\, c : \mu j.k}$$

$$\frac{\begin{array}{c}\Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' \\ \Delta \vdash \mu j.k \ \mathsf{kind} \qquad \Delta, j \vdash k' \ \mathsf{kind}\end{array}}{\begin{array}{c}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c) : \mu j.k \to k'[\mu j.k/j] \\ (j \text{ only positive in } k', \text{ and } j, \alpha, \varphi \notin \Delta)\end{array}}$$

$$\frac{}{\Delta \vdash \mathtt{int} : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta \vdash \tau_1 \to \tau_2 : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta \vdash \tau_1 \times \tau_2 : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 : \mathsf{Type} \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta \vdash \tau_1 + \tau_2 : \mathsf{Type}}$$

$$\frac{\Delta, \alpha{:}k \vdash \tau : \mathsf{Type} \qquad \Delta \vdash k \ \mathsf{kind}}{\Delta \vdash \forall \alpha{:}k.\tau : \mathsf{Type}} \ (\alpha \notin \Delta)$$

$$\frac{\Delta, \alpha{:}k \vdash \tau : \mathsf{Type} \qquad \Delta \vdash k \ \mathsf{kind}}{\Delta \vdash \exists \alpha{:}k.\tau : \mathsf{Type}} \ (\alpha \notin \Delta)$$

$$\Delta \vdash \mathtt{void} : \mathsf{Type}$$

$$\Delta \vdash \mathtt{unit} : \mathsf{Type}$$

$$\frac{\begin{array}{c}\Delta \vdash c : (k \to \mathsf{Type}) \to k \to \mathsf{Type} \\ \Delta \vdash k \ \mathsf{kind} \qquad \Delta \vdash c' : k\end{array}}{\Delta \vdash \mathtt{rec}_k(c, c') : \mathsf{Type}}$$

A.3 Constructor Equivalence

$\boxed{\Delta \vdash c = c' : k}$

$$\frac{\begin{array}{c}\Delta \vdash c' : k[\mu j.k/j] \qquad \Delta, j \vdash k' \ \mathsf{kind} \\ \Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c : k' \quad \Delta \vdash \mu j.k \ \mathsf{kind}\end{array}}{\begin{array}{l}\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)(\mathtt{fold}_{\mu j.k}\, c') = \\ \quad c[\mu j.k, c', \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c)/j, \alpha, \varphi] \\ \qquad : k'[\mu j.k/j] \\ \quad (j \text{ only positive in } k' \text{ and } j, \alpha, \varphi \notin \Delta)\end{array}}$$

$$\frac{\Delta \vdash c_1 : k \qquad \Delta \vdash c_2 : k'}{\Delta \vdash \mathtt{prj}_1 \langle c_1, c_2 \rangle = c_1 : k}$$

$$\frac{\Delta \vdash c_1 : k' \qquad \Delta \vdash c_2 : k}{\Delta \vdash \mathtt{prj}_2 \langle c_1, c_2 \rangle = c_2 : k}$$

$$\frac{\Delta \vdash c : k_1 \times k_2}{\Delta \vdash \langle \mathtt{prj}_1\, c, \mathtt{prj}_2\, c \rangle = c : k_1 \times k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash k' \ \mathsf{kind} \\ \Delta, \alpha{:}k' \vdash c : k' \qquad \Delta \vdash c' : k\end{array}}{\Delta \vdash (\lambda \alpha{:}k'.c)c' = c[c'/\alpha] : k} \ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c : k' \to k}{\Delta \vdash (\lambda \alpha{:}k'.c\alpha) = c : k' \to k} \ (\alpha \notin FV(c))$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}k_1 \vdash c_1 : k \qquad \Delta, \alpha{:}k_2 \vdash c_2 : k \\ \Delta \vdash c : k_1 \qquad \Delta \vdash k_2 \ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_1^{k_1+k_2}\, c, \alpha.c_1, \alpha.c_2) = c_1[c/\alpha] : k}$$

$$\frac{\begin{array}{c}\Delta, \alpha{:}k_1 \vdash c_1 : k \qquad \Delta, \alpha{:}k_2 \vdash c_2 : k \\ \Delta \vdash c : k_2 \qquad \Delta \vdash k_1 \ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{case}(\mathtt{inj}_2^{k_1+k_2}\, c, \alpha.c_1, \alpha.c_2) = c_2[c/\alpha] : k}$$

$$\frac{\Delta \vdash c : k_1 + k_2}{\Delta \vdash \mathtt{case}(c, \alpha_1.\,\mathtt{inj}_1^{k_1+k_2}\,\alpha_1, \alpha_2.\,\mathtt{inj}_2^{k_1+k_2}\,\alpha_2) = c : k_1 + k_2}$$

$$\frac{\Delta \vdash c : k}{\Delta \vdash c = c : k}$$

$$\frac{\Delta \vdash c' = c : k}{\Delta \vdash c = c' : k}$$

$$\frac{\Delta \vdash c_1 = c_2 : k \qquad \Delta \vdash c_2 = c_3 : k}{\Delta \vdash c_1 = c_3 : k}$$

$$\frac{\Delta, \alpha{:}k' \vdash c = c' : k \qquad \Delta \vdash k'\ \mathsf{kind}}{\Delta \vdash \lambda\alpha{:}k'.c = \lambda\alpha{:}k'.c' : k' \to k}\ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c_1 = c_1' : k' \to k \qquad \Delta \vdash c_2 = c_2' : k'}{\Delta \vdash c_1 c_2 = c_1' c_2' : k}$$

$$\frac{\Delta \vdash c_1 = c_1' : k_1 \qquad \Delta \vdash c_2 = c_2' : k_2}{\Delta \vdash \langle c_1, c_2 \rangle = \langle c_1', c_2' \rangle : k_1 \times c_2}$$

$$\frac{\Delta \vdash c = c' : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_1\, c = \mathtt{prj}_1\, c' : k_1}$$

$$\frac{\Delta \vdash c = c' : k_1 \times k_2}{\Delta \vdash \mathtt{prj}_2\, c = \mathtt{prj}_2\, c' : k_2}$$

$$\frac{\Delta \vdash c = c' : k_1 \qquad \Delta \vdash k_2\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_1^{k_1+k_2}\, c = \mathtt{inj}_1^{k_1+k_2}\, c' : k_1 + k_2}$$

$$\frac{\Delta \vdash c = c' : k_2 \qquad \Delta \vdash k_1\ \mathsf{kind}}{\Delta \vdash \mathtt{inj}_2^{k_1+k_2}\, c = \mathtt{inj}_2^{k_1+k_2}\, c' : k_1 + k_2}$$

$$\frac{\begin{array}{c}\Delta \vdash c = c' : k_1 + k_2 \\ \Delta, \alpha{:}k_1 \vdash c_1 = c_1' : k \\ \Delta, \alpha{:}k_2 \vdash c_2 = c_2' : k\end{array}}{\Delta \vdash \mathtt{case}(c, \alpha.c_1, \alpha.c_2) = \mathtt{case}(c', \alpha.c_1', \alpha.c_2') : k}\ (\alpha \notin \Delta)$$

$$\frac{\Delta \vdash c = c' : k[\mu j.k/j]}{\Delta \vdash \mathtt{fold}_{\mu j.k}\, c = \mathtt{fold}_{\mu j.k}\, c' : \mu j.k}$$

$$\frac{\begin{array}{c}\Delta, j, \alpha{:}k, \varphi{:}j \to k' \vdash c_1 = c_2 : k' \\ \Delta \vdash \mu j.k\ \mathsf{kind} \qquad \Delta, j \vdash k'\ \mathsf{kind}\end{array}}{\Delta \vdash \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c_1) = \mathtt{pr}(j, \alpha{:}k, \varphi{:}j \to k'.c_2) : \mu j.k \to k'[\mu j.k/j]}$$
$$(j \text{ only positive in } k' \text{ and } j, \alpha, \varphi \notin \Delta)$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type}}{\Delta \vdash \tau_1 \to \tau_2 = \tau_1' \to \tau_2' : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type}}{\Delta \vdash \tau_1 \times \tau_2 = \tau_1' \times \tau_2' : \mathsf{Type}}$$

$$\frac{\Delta \vdash \tau_1 = \tau_1' : \mathsf{Type} \qquad \Delta \vdash \tau_2 = \tau_2' : \mathsf{Type}}{\Delta \vdash \tau_1 + \tau_2 = \tau_1' + \tau_2' : \mathsf{Type}}$$

$$\frac{\Delta, \alpha{:}k \vdash \tau = \tau' : \mathsf{Type} \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta \vdash \forall\alpha{:}k.\tau = \forall\alpha{:}k.\tau' : \mathsf{Type}}\ (\alpha \notin \Delta)$$

$$\frac{\Delta, \alpha{:}k \vdash \tau = \tau' : \mathsf{Type} \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta \vdash \exists\alpha{:}k.\tau = \exists\alpha{:}k.\tau' : \mathsf{Type}}\ (\alpha \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta \vdash k\ \mathsf{kind} \qquad \Delta \vdash c_2 = c_2' : k \\ \Delta \vdash c_1 = c_1' : (k \to \mathsf{Type}) \to k \to \mathsf{Type}\end{array}}{\Delta \vdash \mathtt{rec}_k(c_1, c_2) = \mathtt{rec}_k(c_1', c_2') : \mathsf{Type}}$$

A.4 Term Formation

$$\boxed{\Delta; \Gamma \vdash e : \tau}$$

$$\frac{}{\Delta; \Gamma \vdash i : \mathtt{int}} \qquad \frac{}{\Delta; \Gamma \vdash * : \mathtt{unit}} \qquad \frac{}{\Delta; \Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Delta; \Gamma, x{:}\tau' \vdash e : \tau \qquad \Delta \vdash \tau' : \mathsf{Type}}{\Delta; \Gamma \vdash \lambda x{:}\tau'.e : \tau' \to \tau}\ (x \notin \Gamma)$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau' \to \tau \qquad \Delta; \Gamma \vdash e_2 : \tau'}{\Delta; \Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \tau_1 \qquad \Delta; \Gamma \vdash e_2 : \tau_2}{\Delta; \Gamma \vdash \langle e_1, e_2 \rangle : \tau_1 \times \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \mathtt{prj}_1\, e : \tau_1}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \times \tau_2}{\Delta; \Gamma \vdash \mathtt{prj}_2\, e : \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_1 \qquad \Delta \vdash \tau_2 : \mathsf{Type}}{\Delta; \Gamma \vdash \mathtt{inj}_1^{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}$$

$$\frac{\Delta; \Gamma \vdash e : \tau_2 \qquad \Delta \vdash \tau_1 : \mathsf{Type}}{\Delta; \Gamma \vdash \mathtt{inj}_2^{\tau_1+\tau_2}\, e : \tau_1 + \tau_2}$$

$$\frac{\begin{array}{c}\Delta; \Gamma \vdash e : \tau_1 + \tau_2 \\ \Delta; \Gamma, x{:}\tau_1 \vdash e_1 : \tau \\ \Delta; \Gamma, x{:}\tau_2 \vdash e_2 : \tau\end{array}}{\Delta; \Gamma \vdash \mathtt{case}(e, x.e_1, x.e_2) : \tau}\ (x \notin \Gamma)$$

$$\frac{\Delta, \alpha{:}k; \Gamma \vdash v : \tau \qquad \Delta \vdash k\ \mathsf{kind}}{\Delta; \Gamma \vdash \Lambda\alpha{:}k.v : \forall\alpha{:}k.\tau}\ (\alpha \notin \Delta)$$

$$\frac{\Delta;\Gamma \vdash e : \forall\alpha{:}k.\tau \qquad \Delta \vdash c' : k}{\Delta;\Gamma \vdash e[c'] : \tau[c'/\alpha]}$$

$$\frac{\Delta;\Gamma, f{:}\tau \vdash e : \tau \qquad \Delta \vdash \tau : \mathsf{Type}}{\Delta;\Gamma \vdash \mathtt{fix}\, f{:}\tau.e : \tau}$$
$$(f \notin \Gamma \text{ and } e = \Lambda\alpha_1{:}k_1 \ldots \Lambda\alpha_n{:}k_n.\lambda x{:}\tau'.e')$$

$$\frac{\begin{array}{c}\Delta,\alpha{:}k \vdash \tau : \mathsf{Type}\\ \Delta \vdash c : k \qquad \Delta;\Gamma \vdash e : \tau[c/\alpha]\end{array}}{\Delta;\Gamma \vdash \mathtt{pack}\, e\, \mathtt{as}\, \exists\alpha{:}k.\tau\ \mathtt{hiding}\ c : \exists\alpha{:}k.\tau}\ (\alpha \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e_1 : \exists\alpha{:}k.\tau_2\\ \Delta,\alpha{:}k;\Gamma,x{:}\tau_2 \vdash e_2 : \tau_1\end{array}}{\Delta;\Gamma \vdash \mathtt{unpack}\,\langle\alpha,x\rangle = e_1\,\mathtt{in}\,e_2 : \tau_1}\ \begin{pmatrix}\alpha \notin \Delta, FV(\tau)\\ x \notin \Gamma\end{pmatrix}$$

$$\frac{\Delta;\Gamma \vdash e : \mathtt{rec}_k(c,c')}{\Delta;\Gamma \vdash \mathtt{unfold}\ e : c(\lambda\alpha{:}k.\,\mathtt{rec}_k(c,\alpha))c'}$$

$$\frac{\begin{array}{c}\Delta;\Gamma \vdash e : c(\lambda\alpha{:}k.\,\mathtt{rec}_k(c,\alpha))c'\\ \Delta \vdash \mathtt{rec}_k(c,c') : \mathsf{Type}\end{array}}{\Delta;\Gamma \vdash \mathtt{fold}_{\mathtt{rec}_k(c,c')}\ e : \mathtt{rec}_k(c,c')}$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\Delta';\Gamma[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad e_1[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] : \tau[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha]\\ \Delta,\beta{:}k_2,\Delta';\Gamma[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad e_2[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] : \tau[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha]\\ \Delta,\alpha{:}k_1+k_2,\Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta,\alpha{:}k_1+k_2,\Delta';\Gamma \vdash \mathtt{ccase}_\tau(c,\beta.e_1,\beta.e_2) : \tau}$$
$$(\beta \notin \Delta)$$

$$\frac{\begin{array}{c}\Delta,\beta{:}k_1,\gamma{:}k_2,\Delta';\Gamma[\langle\beta,\gamma\rangle/\alpha] \vdash e[\langle\beta,\gamma\rangle/\alpha] : \tau[\langle\beta,\gamma\rangle/\alpha]\\ \Delta,\alpha{:}k_1\times k_2,\Delta' \vdash c = \alpha : k_1 \times k_2\end{array}}{\Delta,\alpha{:}k_1\times k_2,\Delta';\Gamma \vdash \mathtt{let}_\tau\langle\beta,\gamma\rangle = c\,\mathtt{in}\,e : \tau}$$
$$(\beta,\gamma \notin \Delta)$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k[\mu j.k/j],\Delta';\Gamma[\mathtt{fold}_{\mu j.k}\ \beta/\alpha] \vdash\\ \qquad e[\mathtt{fold}_{\mu j.k}\ \beta/\alpha] : \tau[\mathtt{fold}_{\mu j.k}\ \beta/\alpha]\\ \Delta,\alpha{:}\mu j.k,\Delta' \vdash c = \alpha : \mu j.k\end{array}}{\Delta,\alpha,\Delta'{:}\mu j.k;\Gamma \vdash \mathtt{let}_\tau(\mathtt{fold}_{\mu j.k}\ \beta) = c\,\mathtt{in}\,e : \tau}\ (\beta \notin \Delta)$$

$$\frac{\Delta \vdash c = \mathtt{inj}_1^{k_1+k_2}\ c' : k_1+k_2 \qquad \Delta;\Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta;\Gamma \vdash \mathtt{ccase}_\tau(c,\alpha.e_1,\alpha.e_2) : \tau}$$

$$\frac{\Delta \vdash c = \mathtt{inj}_2^{k_1+k_2}\ c' : k_1+k_2 \qquad \Delta;\Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta;\Gamma \vdash \mathtt{ccase}_\tau(c,\alpha.e_1,\alpha.e_2) : \tau}$$

$$\frac{\Delta \vdash c = \langle c_1,c_2\rangle : k_1\times k_2 \quad \Delta;\Gamma \vdash e[c_1,c_2/\beta,\gamma] : \tau}{\Delta;\Gamma \vdash \mathtt{let}_\tau\ \langle\beta,\gamma\rangle = c\,\mathtt{in}\,e : \tau}$$

$$\frac{\Delta \vdash c = \mathtt{fold}_{\mu j.k}(c') \quad \Delta;\Gamma \vdash e[c'/\beta] : \tau}{\Delta;\Gamma \vdash \mathtt{let}_\tau\ (\mathtt{fold}_{\mu j.k}\ \beta) = c\,\mathtt{in}\,e : \tau}$$

$$\frac{\Delta;\Gamma \vdash e : \tau' \qquad \Delta \vdash \tau = \tau' : \mathsf{Type}}{\Delta;\Gamma \vdash e : \tau}$$

A.5 Erasure-compatible typing rules (vcase)

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\Delta';\Gamma[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad v[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] : \mathtt{void}\\ \Delta,\beta{:}k_2,\Delta';\Gamma[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad e[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] : \tau[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha]\\ \Delta,\alpha{:}k_1+k_2,\Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta,\alpha{:}k_1+k_2,\Delta';\Gamma \vdash \mathtt{vcase}_\tau(c,\beta.\,\mathtt{dead}\,v,\beta.e) : \tau}$$
$$(\beta \notin \Delta)$$

$$\frac{\begin{array}{l}\Delta,\beta{:}k_1,\Delta';\Gamma[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad e[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha] : \tau[\mathtt{inj}_1^{k_1+k_2}\ \beta/\alpha]\\ \Delta,\beta{:}k_2,\Delta';\Gamma[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] \vdash\\ \qquad v[\mathtt{inj}_2^{k_1+k_2}\ \beta/\alpha] : \mathtt{void}\\ \Delta,\alpha{:}k_1+k_2,\Delta' \vdash c = \alpha : k_1+k_2\end{array}}{\Delta,\alpha{:}k_1+k_2,\Delta';\Gamma \vdash \mathtt{vcase}_\tau(c,\beta.e,\beta.\,\mathtt{dead}\,v) : \tau}$$
$$(\beta \notin \Delta)$$

$$\frac{\Delta \vdash c = \mathtt{inj}_1^{k_1+k_2}\ c' : k_1+k_2 \qquad \Delta;\Gamma \vdash e_1[c'/\alpha] : \tau}{\Delta;\Gamma \vdash \mathtt{vcase}_\tau(c,\alpha.e_1,\alpha.\,\mathtt{dead}\,v) : \tau}$$

$$\frac{\Delta \vdash c = \mathtt{inj}_2^{k_1+k_2}\ c' : k_1+k_2 \qquad \Delta;\Gamma \vdash e_2[c'/\alpha] : \tau}{\Delta;\Gamma \vdash \mathtt{vcase}_\tau(c,\alpha.\,\mathtt{dead}\,v,\alpha.e_2) : \tau}$$

B Operational Semantics

Value syntax

$$\begin{array}{lll} v & ::= & i \mid * \mid \lambda x{:}c.e \mid \langle v_1,v_2\rangle \mid \mathtt{inj}_1^{\tau_1+\tau_2}\ v \mid \mathtt{inj}_2^{\tau_1+\tau_2}\ v\\ & \mid & \Lambda\alpha{:}k.v \mid \mathtt{fix}\ f{:}\tau.v \mid \mathtt{fold}_{\mathtt{rec}_k(c,c')}\ v\\ & \mid & \mathtt{pack}\ v\ \mathtt{as}\ \exists\alpha.c_1\ \mathtt{hiding}\ c_2\\ & \mid & x \mid \mathtt{prj}_1\ v \mid \mathtt{prj}_2\ v\end{array}$$

Evaluation rules

$$(\lambda x{:}c.e)v \mapsto e[v/x]$$

$$\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e_2 \mapsto e_2'}{v e_2 \mapsto v e_2'}$$

$$\mathtt{prj}_1\langle v_1,v_2\rangle \mapsto v_1 \qquad \mathtt{prj}_2\langle v_1,v_2\rangle \mapsto v_2$$

$$\frac{e \mapsto e'}{\mathtt{prj}_1\ e \mapsto \mathtt{prj}_1\ e'} \qquad \frac{e \mapsto e'}{\mathtt{prj}_2\ e \mapsto \mathtt{prj}_2\ e'}$$

$$\frac{e_1 \mapsto e_1'}{\langle e_1,e_2\rangle \mapsto \langle e_1',e_2\rangle} \qquad \frac{e_2 \mapsto e_2'}{\langle v,e_2\rangle \mapsto \langle v,e_2'\rangle}$$

$$\texttt{case}(\texttt{inj}_1^{\tau_1+\tau_2}\, v, x_1.e_1, x_2.e_2) \mapsto e_1[v/x_1]$$

$$\texttt{case}(\texttt{inj}_2^{\tau_1+\tau_2}\, v, x_1.e_1, x_2.e_2) \mapsto e_2[v/x_2]$$

$$\frac{e \mapsto e'}{\texttt{inj}_1^{\tau_1+\tau_2}\, e \mapsto \texttt{inj}_1^{\tau_1+\tau_2}\, e'}$$

$$\frac{e \mapsto e'}{\texttt{inj}_2^{\tau_1+\tau_2}\, e \mapsto \texttt{inj}_2^{\tau_1+\tau_2}\, e'}$$

$$\frac{e \mapsto e'}{\texttt{case}(e, x_1.e_1, x_2.e_2) \mapsto \texttt{case}(e', x_1.e_1, x_2.e_2)}$$

$$\Lambda\alpha{:}k.v[c] \mapsto v[c/\alpha] \qquad \frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

$$\frac{c \text{ normalizes to } \texttt{inj}_1\, c'}{\texttt{ccase}(c, \alpha_1.e_1, \alpha_2.e_2) \mapsto e_1[c'/\alpha_1]}$$

$$\frac{c \text{ normalizes to } \texttt{inj}_2\, c'}{\texttt{ccase}(c, \alpha_1.e_1, \alpha_2.e_2) \mapsto e_2[c'/\alpha_2]}$$

$$\frac{c \text{ normalizes to } \langle c_1, c_2\rangle}{\texttt{let}\,\langle\beta,\gamma\rangle = c\,\texttt{in}\,e \mapsto e[c_1, c_2/\beta,\gamma]}$$

$$\frac{c \text{ normalizes to } \texttt{fold}_{\mu j.k}\, c'}{\texttt{let}\,(\texttt{fold}_{\mu j.k}\,\beta) = c\,\texttt{in}\,e \mapsto e[c'/\beta]}$$

$$(\texttt{fix}\, f{:}\tau.v)v' \mapsto (v[\texttt{fix}\, f{:}\tau.v/f])v'$$

$$(\texttt{fix}\, f{:}\tau.v)[c] \mapsto (v[\texttt{fix}\, f{:}\tau.v/f])[c]$$

$$\frac{e \mapsto e'}{\texttt{pack}\, e\, \texttt{as}\, \exists\beta.c_1\ \texttt{hiding}\, c_2 \mapsto \texttt{pack}\, e'\, \texttt{as}\, \exists\beta.c_1\ \texttt{hiding}\, c_2}$$

$$\frac{e \mapsto e'}{\texttt{unpack}\,\langle\alpha, x\rangle = e\,\texttt{in}\,e_2 \mapsto \texttt{unpack}\,\langle\alpha, x\rangle = e'\,\texttt{in}\,e_2}$$

$$\texttt{unfold}\ (\texttt{fold}_{\texttt{rec}_k(c,c')}\, v) \mapsto v$$

$$\frac{e \mapsto e'}{\texttt{fold}_{\texttt{rec}_k(c,c')}\, e \mapsto \texttt{fold}_{\texttt{rec}_k(c,c')}\, e'}$$

$$\frac{e \mapsto e'}{\texttt{unfold}\ e \mapsto \texttt{unfold}\ e'}$$

B.1 Erasure-compatible operational rules (vcase)

Value syntax

$$v ::= \ldots \mid v[c]$$

Evaluation rules Delete evaluation rules for `fix` and `ccase` and replace them with the following:

$$(\texttt{fix}\, f{:}\tau.v)[c_1]\cdots[c_n]v' \mapsto (v[\texttt{fix}\, f{:}\tau.v/f])[c_1]\cdots[c_n]v'$$

$$\frac{c \text{ normalizes to } \texttt{inj}_1\, c'}{\texttt{vcase}(c, \alpha_1.e_1, \alpha_2.\,\texttt{dead}\, v) \mapsto e_1[c'/\alpha_1]}$$

$$\frac{c \text{ normalizes to } \texttt{inj}_2\, c'}{\texttt{vcase}(c, \alpha_1.\,\texttt{dead}\, v, \alpha_2.e_2) \mapsto e_2[c'/\alpha_2]}$$

C Type Erasure Formulation

Although the formal static and operational semantics for the erasable version of LX (Section 5) are for a typed language, we would like to emphasize the point that types are unnecessary for computation and can be safely erased. To do this we exhibit an untyped language, LX°, a translation from LX through type erasure, and the following theorem, which states that execution in the untyped language mirrors execution in the typed language:

Theorem C.1 *1. If $e_1 \mapsto^* e_2$ then $e_1{}^\circ \mapsto^* e_2{}^\circ$.*

2. If $\emptyset \vdash e_1 : \tau$ and $e_1{}^\circ \mapsto^* u$ then there exists e_2 such that $e_1 \mapsto^* e_2$ and $e_2{}^\circ = u$.

From this theorem and the type safety of LX it follows that our untyped semantics is safe.

Corollary C.2 *If $\emptyset \vdash e : \tau$ and $e^\circ \mapsto^* u$ then u is not stuck.*

C.1 Syntax of Untyped Calculus

$$\begin{array}{lll}
(\mathit{terms}) & u & ::= * \mid i \mid x \mid \lambda x.u \mid \texttt{fix}\, f.w \mid u_1 u_2 \\
 & & \quad \mid \langle u_1, u_2\rangle \mid \texttt{prj}_1\, u \mid \texttt{prj}_2\, u \mid \texttt{inj}_1\, u \\
 & & \quad \mid \texttt{inj}_2\, u \mid \texttt{case}(u, x_1.u_1, x_2.u_2) \\
(\mathit{values}) & w & ::= x \mid i \mid \lambda x.u \mid \texttt{fix}\, f.w \mid \langle w_1, w_2\rangle \\
 & & \quad \mid \texttt{inj}_1\, w \mid \texttt{inj}_2\, w \mid \texttt{prj}_1\, w \mid \texttt{prj}_2\, w
\end{array}$$

C.2 Type Erasure

$$
\begin{array}{rcl}
x^\circ & = & x \\
i^\circ & = & i \\
\langle e_1, e_2 \rangle^\circ & = & \langle {e_1}^\circ, {e_2}^\circ \rangle \\
(\mathtt{prj}_i\, e)^\circ & = & \mathtt{prj}_i\, e^\circ \\
(\lambda x{:}\tau.e)^\circ & = & \lambda x.e^\circ \\
(\Lambda\alpha{:}k.v)^\circ & = & v^\circ \\
(\mathtt{fix}\, f{:}\tau.e)^\circ & = & \mathtt{fix}\, f.e^\circ \\
(e_1 e_2)^\circ & = & {e_1}^\circ {e_2}^\circ \\
e[c]^\circ & = & e^\circ \\
\mathtt{pack}\, e\, \mathtt{as}\, c\, \mathtt{hiding}\, {c'}^\circ & = & e^\circ \\
\mathtt{unpack}\, \langle \alpha, x \rangle = e_1\, \mathtt{in}\, {e_2}^\circ & = & (\lambda x.{e_2}^\circ)\, {e_1}^\circ \\
\mathtt{inj}_i^{\tau_1+\tau_2}\, e^\circ & = & \mathtt{inj}_i\, e^\circ \\
\mathtt{case}(e, x_1.e_1, x_2.e_2)^\circ & = & \mathtt{case}(e^\circ, \\
 & & \quad x_1.{e_1}^\circ, x_2.{e_2}^\circ) \\
*^\circ & = & * \\
\mathtt{fold}_{\mathtt{rec}_k(c,c')}\, e^\circ & = & e^\circ \\
\mathtt{unfold}\ e^\circ & = & e^\circ \\
\mathtt{vcase}(c, \alpha_1.e, \alpha_2.\,\mathtt{dead}\, v)^\circ & = & e^\circ \\
\mathtt{vcase}(c, \alpha_1.\,\mathtt{dead}\, v, \alpha_2.e)^\circ & = & e^\circ
\end{array}
$$

C.3 Operational Semantics of LX°

$$(\lambda x.u)w \mapsto u[w/x]$$

$$(\mathtt{fix}\ f.w)w' \mapsto (w[\mathtt{fix}\, f.w/f])w'$$

$$\frac{u_1 \mapsto u_1'}{u_1 u_2 \mapsto u_1' u_2} \qquad \frac{u \mapsto u'}{wu \mapsto wu'}$$

$$\mathtt{prj}_1 \langle w_1, w_2 \rangle \mapsto w_1 \qquad \mathtt{prj}_2 \langle w_1, w_2 \rangle \mapsto w_2$$

$$\frac{u_1 \mapsto u_1'}{\langle u_1, u_2 \rangle \mapsto \langle u_1', u_2 \rangle} \qquad \frac{u \mapsto u'}{\langle w, u \rangle \mapsto \langle w, u' \rangle}$$

$$\frac{u \mapsto u'}{\mathtt{prj}_1\, u \mapsto \mathtt{prj}_1\, u'} \qquad \frac{u \mapsto u'}{\mathtt{prj}_2\, u \mapsto \mathtt{prj}_2\, u'}$$

$$\mathtt{case}(\mathtt{inj}_1\, w, x_1.u_1, x_2.u_2) \mapsto u_1[w/x_1]$$

$$\mathtt{case}(\mathtt{inj}_2\, w, x_1.u_1, x_2.u_2) \mapsto u_2[w/x_2]$$

$$\frac{u \mapsto u'}{\mathtt{inj}_1\, u \mapsto \mathtt{inj}_1\, u'} \qquad \frac{u \mapsto u'}{\mathtt{inj}_2\, u \mapsto \mathtt{inj}_2\, u'}$$

$$\frac{u \mapsto u'}{\mathtt{case}(u, x_1.u_1, x_2.u_2) \mapsto \mathtt{case}(u', x_1.u_1, x_2.u_2)}$$