

# Encoding Intensional Type Analysis

Stephanie Weirich\*

Department of Computer Science, Cornell University  
Ithaca, NY 14850  
sweirich@cs.cornell.edu

**Abstract.** Languages for intensional type analysis permit ad-hoc polymorphism, or run-time analysis of types. However, such languages require complex, specialized constructs to support this operation, which hinder optimization and complicate the meta-theory of these languages. In this paper, we observe that such specialized operators need not be intrinsic to the language, and in fact, their operation may be simulated through standard encodings of iteration in the polymorphic lambda calculus. Therefore, we may more easily add intensional analysis operators to complicated languages via translation, instead of language extension.

## 1 Introduction

Consider a well-known inductive datatype (presented in Standard ML syntax [14] augmented with explicit polymorphism):

```
datatype Tree = Leaf | Node of Tree * Tree
Treerec : ∀a. Tree -> a -> ( a * a -> a ) -> a
```

`Leaf` and `Node` are introduction forms, used to create elements of type `Tree`. The function `Treerec` is an elimination form, iterating computation over an element of type `Tree`, creating a fold or a catamorphism. It accepts a base case (of type `a`) for the leaves and an inductive case (of type `a * a -> a`) for the nodes. For example, we may use `Treerec` to define a function to display a `Tree`. First, we explicitly instantiate the return type `a` with `[string]`. For the leaves, we provide the string `"Leaf"`, and for the nodes we concatenate (with the infix operator `^`) the strings of the subtrees.

```
val showTree = fn x : Tree =>
  Treerec [string] x
    "Leaf"
    (fn (s1:string, s2:string) => "Node(" ^ s1 ^ ", " ^ s2 ^ ")")
```

---

\* This paper is based on work supported in part by the National Science Foundation under Grant No. CCR-9875536. Any opinions, findings and conclusions or recommendations expressed in this publication are those of the author and do not reflect the views of this agency.

As `Tree` is an inductive datatype, it is well known how to encode it in the polymorphic lambda calculus [1]. The basic idea is to encode a `Tree` as its elimination form — a function that iterates over the tree. In other words, a `Leaf` is a function that accepts a base case and an inductive case and returns the base case. Because we do not wish to constrain the return type of iteration, we abstract it, explicitly binding it with  $\lambda a$ .

```
val Leaf =  $\lambda a$ . fn base:a => fn ind:a * a -> a => base
```

Likewise, a `Node`, with two subtrees `x` and `y`, selects the inductive case, passing it the result of continuing the iteration through the two subtrees.

```
val Node (x:Tree) (y:Tree) =
   $\lambda a$ . fn base:a => fn ind:a * a -> a =>
    ind (Treerec [a] x base ind) (Treerec [a] y base ind)
```

However, as all of the iteration is encoded into the data structure itself, the elimination form only needs to pass it on.

```
val Treerec =  $\lambda a$ . fn x : Tree => fn base : a =>
  fn ind : a * a -> a => x [a] base ind
```

Consequently, we may write `Node` more simply as

```
val Node (x:Tree) (y:Tree) =
   $\lambda a$ . fn base:a => fn ind:a * a -> a =>
    ind (x [a] base ind) (y [a] base ind)
```

Now consider another inductive datatype:

```
datatype Type = Int | Arrow of Type * Type
Typerec :  $\forall a$ . Type -> a -> ( a * a -> a ) -> a
```

Ok, so we just changed the names. However, this datatype (or at least the introductory forms of it) is quite common in typed programming languages. It is the inductive definition of the types of the simply-typed lambda calculus.

$$\tau ::= \text{int} \mid \tau \rightarrow \tau$$

Just as we may write functions in ML to create and manipulate `Trees`, in some languages, we may write functions (or *type constructors*) that create and manipulate `Types`. These functions over `Types` must themselves be typed (we use the word *kind* for the types of types). If we use `Type` (notated by  $\Omega$ ) as the base kind, we get what is starting to look like the syntax of the kinds and type constructors of Girard's language  $F_\omega$  [8].

$$\begin{array}{ll} \text{(kinds)} & \kappa ::= \Omega \mid \kappa \rightarrow \kappa \\ \text{(type constructors)} & \tau ::= \text{int} \mid \tau \rightarrow \tau \mid \alpha \mid \lambda \alpha : \kappa . \tau \mid \tau \tau \end{array}$$

The language  $\lambda_i^{ML}$  [9] adds the elimination form *Typerec* to this type constructor language. Because *Typerec* may determine the *structure* of an abstract

---

( <i>kinds</i> )	$\kappa ::= \Omega \mid \kappa_1 \rightarrow \kappa_2$
( <i>type constructors</i> )	$c, \tau ::= \alpha \mid \lambda \alpha : \kappa . c \mid c_1 c_2 \mid \text{int} \mid \tau_1 \rightarrow \tau_2 \mid$ $\mid \text{Typerec}[\kappa] \tau c_i c_{\rightarrow}$
( <i>types</i> )	$\sigma ::= T(\tau) \mid R(\tau) \mid \sigma_1 \rightarrow \sigma_2 \mid \forall \alpha : \kappa . \sigma$
( <i>terms</i> )	$e ::= i \mid x \mid \lambda x : \sigma . e \mid e_1 e_2$ $\mid \Lambda \alpha : \kappa . e \mid e[c] \mid R_i \mid R_{\rightarrow}$ $\mid \text{typerec}[c] e e_i e_{\rightarrow}$

---

**Fig. 1.** Syntax of of the source language,  $\lambda_R$

---

type, its operation is called *intensional analysis*. Furthermore,  $\lambda_i^{ML}$  also allows the definition of a fold over a **Type** to create a *term*, with the special term *typerec*. With this term,  $\lambda_i^{ML}$  supports run-time type analysis, as the identities of type constructors affect run-time execution. For example, just as we defined a function to print out trees, we can define a function to print out types at run time.

```
val showType =  $\Lambda \alpha : \Omega .$ 
  typerec [string] a
    "int"
    (fn (s1:string, s2:string) => "(" ^ s1 ^ " -> " ^ s2 ^ ")")
```

Even though the type constructor *Typerec* and the term *typerec* are very specialized operators in  $\lambda_i^{ML}$ , they are just folds over an inductive data structure. And just as we can encode folds over **Trees** in the polymorphic lambda calculus, we can encode folds over **Types**. Note that to encode the type constructor *Typerec*, we will need to add kind polymorphism to the type constructor language.

In the rest of this paper, we will demonstrate how to encode a language with intensional type analysis operators into a variant of  $F_{\omega}$  augmented with kind polymorphism. The fact that such an encoding exists means that the specialized operators *typerec* and *Typerec* do not need to be an intrinsic part of a programming language for it to support intensional type analysis. Therefore, we may more easily add these operators to complicated languages via a translation semantics, instead of through language extension.

The rest of the paper is organized as follows. Formal descriptions of the source and target languages appear in Section 2, and we present the embedding between them in Section 3. Section 4 describes the limitations of the translation and discusses when one might want an explicit iteration operator in the target language. Section 5 discusses related work and concludes.

Constructor <i>Typerec</i>	Term <i>typerec</i>
<i>Formation</i>	<i>Formation</i>
$\frac{\Delta \vdash \tau : \Omega \quad \Delta \vdash c_i : \kappa}{\Delta \vdash c_{\rightarrow} : \kappa \rightarrow \kappa \rightarrow \kappa}$	$\frac{\Delta; \Gamma \vdash c : \Omega \rightarrow \Omega \quad \Delta; \Gamma \vdash e : R(\tau) \quad \Delta; \Gamma \vdash e_i : T(c(int))}{\Delta; \Gamma \vdash e_{\rightarrow} : \forall \alpha: \Omega. \forall \beta: \Omega. T(c\alpha \rightarrow c\beta \rightarrow c(\alpha \rightarrow \beta))}$
$\Delta \vdash Typerec[\kappa] \tau c_i c_{\rightarrow} : \kappa$	$\frac{\Delta; \Gamma \vdash e_{\rightarrow} : \forall \alpha: \Omega. \forall \beta: \Omega. T(c\alpha \rightarrow c\beta \rightarrow c(\alpha \rightarrow \beta))}{\Delta; \Gamma \vdash typerec[c] e e_i e_{\rightarrow} : T(c\tau)}$
<i>Constructor equivalence</i>	<i>Operational semantics</i>
$\Delta \vdash Typerec[\kappa] int c_i c_{\rightarrow} = c_i : k$	$typerec[c] R_i e_i e_{\rightarrow} \mapsto e_i$
$\Delta \vdash Typerec[\kappa](\tau_1 \rightarrow \tau_2) c_i c_{\rightarrow} =$ $c_{\rightarrow} (Typerec[\kappa] \tau_1 c_i c_{\rightarrow})$ $(Typerec[\kappa] \tau_2 c_i c_{\rightarrow}) : k$	$typerec[c] (R_{\rightarrow}[\tau_1][\tau_2] v_1 v_2) e_i e_{\rightarrow} \mapsto$ $e_{\rightarrow} [\tau_1][\tau_2] (typerec[c] v_1 e_i e_{\rightarrow})$ $(typerec[c] v_2 e_i e_{\rightarrow})$

**Fig. 2.** *Typerec* and *typerec*

## 2 The Languages

Instead of directly presenting a translation of  $\lambda_i^{ML}$ , we instead choose as the source language Cray *et al.*'s  $\lambda_R$  [5]. Because we will define two elimination forms, *typerec* and *Typerec*, we will need to separate type information used at the term level for run-time type analysis from that used at the type constructor level for static type checking. The language  $\lambda_R$  exhibits this separation by using terms that *represent* type constructors for analysis at run time, reserving type constructors for type-level analysis. A translation from  $\lambda_i^{ML}$  into  $\lambda_R$  provides term representations (suitable for *typerec*) for each type constructor abstracted by the source program.

To avoid analyzing quantified types, the core of  $\lambda_R$  is a predicative variant of  $F_{\omega}$ . The quantifier  $\forall \alpha: \kappa. \sigma$  ranges only over “small” types which do not include the quantified types. Therefore, the syntax (Figure 1) is divided into four syntactic categories: type *constructors* described by *kinds*, and *terms* described by *types*. By convention we use the meta-variable  $\tau$  for constructors of kind  $\Omega$  (those equivalent to unquantified types) and  $c$  for arbitrary constructors. A constructor  $\tau$  of kind  $\Omega$  may be explicitly coerced to a type with  $T(\tau)$ .

The semantics of  $\lambda_R$  includes judgments for type constructor formation  $\Delta \vdash c : k$ , type constructor equality  $\Delta \vdash c_1 = c_2 : k$ , type formation  $\Delta \vdash \sigma$ , type equality  $\Delta \vdash \sigma_1 = \sigma_2$ , term formation  $\Delta; \Gamma \vdash e : \sigma$  and small-step operational semantics  $e \mapsto e'$ . In these judgments,  $\Delta$  and  $\Gamma$  are contexts describing the kinds and types of the free constructor and term variables.

The semantics of the type constructor *Typerec* and term *typerec* appears in Figure 2. Unlike  $\lambda_i^{ML}$ , the argument to *typerec* is a term representing a type constructor, not the type constructor itself. The type  $R(\tau)$  describes such a

term representing  $\tau$ . The type is singular; for any  $\tau$ , only one term inhabits  $R(\tau)$ . Therefore, once the identity of a term of type  $R(\tau)$  is determined, so is the identity of  $\tau$ . For example, if  $x : R(\alpha)$  and  $x$  matches the representation of the type *int*, denoted  $R_i$ , then we know  $\alpha$  must be *int*.

Arrow types in  $\lambda_R$  are represented by the  $R_{\rightarrow}$  term. This term requires the two types of the subcomponents of the arrow type and the two terms representing those types.

$$R_{\rightarrow} : \forall\alpha:\Omega.\forall\beta:\Omega. R(\alpha) \rightarrow R(\beta) \rightarrow R(\alpha \rightarrow \beta)$$

For example, the type *int*  $\rightarrow$  *int* is represented by the term

$$R_{\rightarrow}[int][int] R_i R_i$$

One extremely useful property of *typerec* not illustrated by the `showType` example from Section 1, is that the types of the  $e_i$  and  $e_{\rightarrow}$  branches to *typerec* may depend on the identity of the analyzed type. If the argument to *typerec* is a term of type  $R(\tau)$ , the result type of the expression is  $T(c\tau)$ , where  $c$  may be an arbitrary type constructor. (The *typerec* term is annotated by  $c$  to permit syntax-directed type checking.) However, instead of requiring that the  $e_i$  be of type  $T(c\tau)$ , it may be of type  $T(c\textit{int})$ , reflecting the fact that in  $e_i$  branch we know  $\tau$  is *int*. Likewise, the return type of the  $e_{\rightarrow}$  is  $T(c(\alpha \rightarrow \beta))$ , for some  $\alpha$  and  $\beta$ .

There are several differences between  $\lambda_R$  presented in this paper and the language of Crary *et al.* [5]. To simplify presentation, this version is call-by-name instead of call-by-value. Also, here the result of *typerec* is annotated with a type constructor, instead of a type. However, we make two essential changes to support the embedding presented in this paper. First, we prevent  $R$ -types from appearing as an argument to *typerec* or *Typerec*, by making  $R$  a part of the type language, and not a type constructor. We discuss in the next section why this restriction is necessary.

Second, although *typerec* and *Typerec* usually define a primitive recursive fold over kind  $\Omega$  (also called a *paramorphism* [12,11]), in this language we replace these operators with their iterative cousins (which define *catamorphisms*). The difference between iteration and primitive recursion is apparent in the kind of  $c_{\rightarrow}$  and the type of  $e_{\rightarrow}$ . With primitive recursion, the arrow branch receives four arguments: the two subcomponents of the arrow constructor and two results of continuing the fold through these subcomponents. In iteration, on the other hand, the arrow branch receives only two arguments, the results of the continued fold.<sup>1</sup> We discuss this restriction further in Section 4.1.

The remainder of the static and operational semantics for this language, and for the primitive recursive versions, *typerec<sup>PR</sup>* and *Typerec<sup>PR</sup>*, appear in Appendices A.1 and B. For space reasons, we omit the formation rules for types and type constructors, as they may be inferred from the rules for equality.

<sup>1</sup> Because we cannot separate type constructors passed for static type checking, from those passed for dynamic type analysis in  $\lambda_i^{ML}$ , we *must* provide the subcomponents of the arrow type to the arrow branch of *typerec*. Therefore, we cannot define an iterative version of *typerec* for that language.

$$\begin{aligned}
 (\text{kinds}) \quad \kappa &:: = \Omega \mid \kappa_1 \rightarrow \kappa_2 \mid \chi \mid \forall \chi. \kappa \\
 (\text{con's}) \quad c, \tau &:: = \alpha \mid \lambda \alpha: \kappa. c \mid c_1 c_2 \mid \Lambda \chi. c \mid c[\kappa] \\
 &\quad \mid \text{int} \mid \tau_1 \rightarrow \tau_2 \mid \forall \alpha: \kappa. \tau \\
 (\text{terms}) \quad e &:: = i \mid x \mid \lambda x: \tau. e \mid e_1 e_2 \\
 &\quad \mid \Lambda \alpha: \kappa. e \mid e[c]
 \end{aligned}$$

**Fig. 3.** Syntax of the target language,  $\lambda U^-$

The target language of the translation is  $\lambda U^-$ , the language  $F_\omega$  augmented with kind polymorphism at the type constructor level (Figure 3). As the target language is impredicative, both types and type constructors are in the same syntactic class. In Section 4.2 we discuss why we might want alternate target languages not based on impredicative polymorphism. The static and operational semantics of  $\lambda U^-$  appear in Appendices A.2 and C.

### 3 The Translation

The translation of  $\lambda_R$  into  $\lambda U^-$  can be thought of as two separate translations: A translation of the kinds and constructors of  $\lambda_R$  into the kinds and constructors of  $\lambda U^-$  and a translation of the types and terms of  $\lambda_R$  into the constructors and terms of  $\lambda U^-$ . For reference, the complete translation appears in Figure 4.

#### 3.1 Defining Iteration

To define the translation of *Typerec* we use the traditional encoding of inductive datatypes in impredicative polymorphism. As before, we encode  $\tau$ , of kind  $\Omega$  as its elimination form: a function that chooses between two given branches — one for  $c_i$ , one for  $c_{\rightarrow}$ . Then *Typerec* $[\kappa] \tau c_i c_{\rightarrow}$  can be implemented with

$$[[\tau]][[\kappa]] [c_i] [c_{\rightarrow}]$$

As  $\tau$  is of kind type, we define  $[[\Omega]]$  to reflect the fact that  $[[\tau]]$  must accept an arbitrary kind and the two branches.

$$[[\Omega]] = \forall \chi. \chi \rightarrow (\chi \rightarrow \chi \rightarrow \chi) \rightarrow \chi$$

Accordingly, the encoding of the type constructor *int* just returns its first argument (the kinds of the arguments have been elided)

$$[[\text{int}]] = (\Lambda \chi. \lambda t. \lambda \alpha. t)$$

Now consider the constructor equality rule when the argument to *Typerec* is an arrow type. The translation of the arrow type constructor  $\rightarrow$ , should apply

the second argument (the  $c_{\rightarrow}$  branch) to the result of continuing the recursion through the two subcomponents.

$$\llbracket \tau_1 \rightarrow \tau_2 \rrbracket = \Lambda\chi.\lambda\iota.\lambda\alpha.\alpha(\llbracket \tau_1 \rrbracket[\chi] \iota \alpha)(\llbracket \tau_2 \rrbracket[\chi] \iota \alpha)$$

A critical property of this translation is that it preserve the equivalences that exist in the source language. For example, one equivalence we must preserve from the source language is that

$$\llbracket \text{Typerec}[\kappa] (\tau_1 \rightarrow \tau_2) c_i c_{\rightarrow} \rrbracket = \llbracket c_{\rightarrow}(\text{Typerec}[\kappa] \tau_1 c_i c_{\rightarrow})(\text{Typerec}[\kappa] \tau_2 c_i c_{\rightarrow}) \rrbracket$$

If we expand the left side, we get

$$(\Lambda\chi.\lambda\iota.\lambda\alpha.\alpha(\llbracket \tau_1 \rrbracket[\chi] \iota \alpha)(\llbracket \tau_2 \rrbracket[\chi] \iota \alpha)) \llbracket [\kappa] \rrbracket \llbracket [c_i] \rrbracket \llbracket [c_{\rightarrow}] \rrbracket$$

This term is then  $\beta$ -equivalent to the expansion of the right hand side.

$$\llbracket [c_{\rightarrow}] \rrbracket (\llbracket \tau_1 \rrbracket \llbracket [\kappa] \rrbracket \llbracket [c_i] \rrbracket \llbracket [c_{\rightarrow}] \rrbracket) (\llbracket \tau_2 \rrbracket \llbracket [\kappa] \rrbracket \llbracket [c_i] \rrbracket \llbracket [c_{\rightarrow}] \rrbracket)$$

Because type constructors are a separate syntactic class from types, we must define  $\llbracket T(\tau) \rrbracket$ , the coercion between them. We convert  $\llbracket \tau \rrbracket$  of kind  $\llbracket \Omega \rrbracket$  into a  $\lambda U^-$  constructor of kind  $\Omega$  using the iteration built into  $\llbracket \tau \rrbracket$ .

$$\llbracket T(\tau) \rrbracket = \llbracket \tau \rrbracket \llbracket \Omega \rrbracket \text{ int } (\lambda\alpha:\Omega.\lambda\beta:\Omega.\alpha \rightarrow \beta)$$

For example,

$$\begin{aligned} \llbracket T(\text{int}) \rrbracket &= \llbracket \text{int} \rrbracket \llbracket \Omega \rrbracket \text{ int } (\lambda\alpha:\Omega.\lambda\beta:\Omega.\alpha \rightarrow \beta) \\ &= (\Lambda\chi.\lambda\iota.\lambda\alpha.\iota) \llbracket \Omega \rrbracket \text{ int } (\lambda\alpha:\Omega.\lambda\beta:\Omega.\alpha \rightarrow \beta) \\ &=_{\beta} \text{int} \end{aligned}$$

We use a very similar encoding for *typerec* at the term level, as we do for *Typerec*. Again, we wish to apply the translation of the argument to the translation of the branches, and let the argument select between them.

$$\llbracket \text{typerec}[c]e e_i e_{\rightarrow} \rrbracket \quad \text{as} \quad \llbracket [e] \rrbracket \llbracket [c] \rrbracket \llbracket [e_i] \rrbracket \llbracket [e_{\rightarrow}] \rrbracket$$

The translations of  $R_i$  and  $R_{\rightarrow}$  are analogous to those of the type constructors *int* and  $\rightarrow$ . However, there is a subtle point about the definition of  $R(\tau)$ , the type of the argument to *typerec*. To preserve typing, we define  $\llbracket R(\tau) \rrbracket$  as:

$$\begin{aligned} \forall\gamma:\llbracket \Omega \rightarrow \Omega \rrbracket. \llbracket T(\gamma \text{ int}) \rrbracket \\ \rightarrow \llbracket \forall\alpha:\Omega.\forall\beta:\Omega. T(\gamma\alpha) \rightarrow T(\gamma\beta) \rightarrow T(\gamma(\alpha \rightarrow \beta)) \rrbracket \\ \rightarrow \llbracket T(\gamma\tau) \rrbracket \end{aligned}$$

Here we see why  $R$  cannot be a type constructor; if it were, we would have an additional branch for it in the translation of  $T$  mapping the  $R$  constructor to the  $R$  type. So the definition would be

$$\llbracket T(\tau) \rrbracket = \llbracket \tau \rrbracket \llbracket \Omega \rrbracket \text{ int } (\lambda\alpha:\Omega.\lambda\beta:\Omega.\alpha \rightarrow \beta) (\lambda\alpha:\Omega. R(\alpha)) \quad (\text{WRONG})$$

causing the definition of  $\llbracket R(\tau) \rrbracket$  to be recursive.

*Kind Translation*

$$\begin{aligned} \llbracket \Omega \rrbracket &= \forall \chi. \chi \rightarrow (\chi \rightarrow \chi \rightarrow \chi) \rightarrow \chi \\ \llbracket \kappa_1 \rightarrow \kappa_2 \rrbracket &= \llbracket \kappa_1 \rrbracket \rightarrow \llbracket \kappa_2 \rrbracket \end{aligned}$$

*Constructor Translation*

$$\begin{aligned} \llbracket \alpha \rrbracket &= \alpha \\ \llbracket \lambda \alpha : \kappa. c \rrbracket &= \lambda \alpha : \llbracket \kappa \rrbracket. \llbracket c \rrbracket \\ \llbracket c_1 c_2 \rrbracket &= \llbracket c_1 \rrbracket \llbracket c_2 \rrbracket \\ \llbracket int \rrbracket &= \Lambda \chi. \lambda \iota : \chi. \lambda \alpha : \chi \rightarrow \chi \rightarrow \chi. \iota \\ \llbracket \tau_1 \rightarrow \tau_2 \rrbracket &= \Lambda \chi. \lambda \iota : \chi. \lambda \alpha : \chi \rightarrow \chi \rightarrow \chi. \\ &\quad \alpha (\llbracket \tau_1 \rrbracket [\chi] \iota \alpha) (\llbracket \tau_2 \rrbracket [\chi] \iota \alpha) \\ \llbracket Typerec[\kappa] \tau c_i c_{\rightarrow} \rrbracket &= \llbracket \tau \rrbracket \llbracket \llbracket \kappa \rrbracket \rrbracket \llbracket c_i \rrbracket \llbracket c_{\rightarrow} \rrbracket \end{aligned}$$

*Type Translation*

$$\begin{aligned} \llbracket T(\tau) \rrbracket &= \llbracket \tau \rrbracket \llbracket \Omega \rrbracket int (\lambda \alpha : \Omega. \lambda \beta : \Omega. \alpha \rightarrow \beta) \\ \llbracket R(\tau) \rrbracket &= \forall \gamma : \llbracket \Omega \rightarrow \Omega \rrbracket. \llbracket T(\gamma int) \rrbracket \\ &\quad \rightarrow \llbracket \forall \alpha : \Omega. \forall \beta : \Omega. T(\gamma \alpha) \rightarrow T(\gamma \beta) \rightarrow T(\gamma(\alpha \rightarrow \beta)) \rrbracket \\ &\quad \rightarrow \llbracket T(\gamma \tau) \rrbracket \\ \llbracket int \rrbracket &= int \\ \llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket &= \llbracket \sigma_1 \rrbracket \rightarrow \llbracket \sigma_2 \rrbracket \\ \llbracket \forall \alpha : \kappa. \sigma \rrbracket &= \forall \alpha : \llbracket \kappa \rrbracket. \llbracket \sigma \rrbracket \end{aligned}$$

*Term Translation*

$$\begin{aligned} \llbracket x \rrbracket &= x \\ \llbracket \lambda x : \sigma. e \rrbracket &= \lambda x : \llbracket \sigma \rrbracket. \llbracket e \rrbracket \\ \llbracket e_1 e_2 \rrbracket &= \llbracket e_1 \rrbracket \llbracket e_2 \rrbracket \\ \llbracket \Lambda \alpha : \kappa. e \rrbracket &= \Lambda \alpha : \llbracket \kappa \rrbracket. \llbracket e \rrbracket \\ \llbracket e[c] \rrbracket &= \llbracket e \rrbracket \llbracket \llbracket c \rrbracket \rrbracket \\ \llbracket R_i \rrbracket &= (\Lambda \gamma : \llbracket \Omega \rightarrow \Omega \rrbracket. \lambda i : \llbracket T(\gamma int) \rrbracket. \\ &\quad \lambda a : \llbracket \forall \alpha : \Omega. \forall \beta : \Omega. T(\gamma \alpha) \rightarrow T(\gamma \beta) \rightarrow T(\gamma(\alpha \rightarrow \beta)) \rrbracket. i) \\ \llbracket R_{\rightarrow} \rrbracket &= \Lambda \alpha : \llbracket \Omega \rrbracket. \Lambda \beta : \llbracket \llbracket \Omega \rrbracket \rrbracket. \lambda x_1 : \llbracket R(\alpha) \rrbracket. \lambda x_2 : \llbracket R(\beta) \rrbracket \\ &\quad (\Lambda \gamma : \llbracket \Omega \rightarrow \Omega \rrbracket. \lambda i : \llbracket T(\gamma int) \rrbracket. \\ &\quad \lambda a : \llbracket \forall \alpha : \Omega. \forall \beta : \Omega. T(\gamma \alpha) \rightarrow T(\gamma \beta) \rightarrow T(\gamma(\alpha \rightarrow \beta)) \rrbracket. \\ &\quad a [\alpha] \llbracket \beta \rrbracket (x_1 [\gamma] i a) (x_2 [\gamma] i a)) \\ \llbracket typerec[c] e e_i e_{\rightarrow} \rrbracket &= \llbracket e \rrbracket \llbracket \llbracket c \rrbracket \rrbracket \llbracket e_i \rrbracket \llbracket e_{\rightarrow} \rrbracket \end{aligned}$$

**Fig. 4.** Translation of  $\lambda_R$  into  $\lambda U^-$



### 3.2 Properties of the Embedding

The translation presented above enjoys the following properties. Define  $\llbracket \Delta \rrbracket$  as  $\{\alpha: \llbracket \Delta(\alpha) \rrbracket \mid \alpha \in \text{Dom}(\Delta)\}$  and  $\llbracket \Gamma \rrbracket$  as  $\{x: \llbracket \Gamma(x) \rrbracket \mid x \in \text{Dom}(\Gamma)\}$ .

**Theorem 1 (Static Correctness).**

1.  $\emptyset \vdash \llbracket \kappa \rrbracket$
2. If  $\Delta \vdash c : \kappa$  then  $\llbracket \Delta \rrbracket \vdash \llbracket c \rrbracket : \llbracket \kappa \rrbracket$ .
3. If  $\Delta \vdash c = c' : \kappa$  then  $\llbracket \Delta \rrbracket \vdash \llbracket c \rrbracket = \llbracket c' \rrbracket : \llbracket \kappa \rrbracket$ .
4. If  $\Delta \vdash \sigma$  then  $\llbracket \Delta \rrbracket \vdash \llbracket \sigma \rrbracket : \Omega$ .
5. If  $\Delta \vdash \sigma = \sigma'$  then  $\llbracket \Delta \rrbracket \vdash \llbracket \sigma \rrbracket = \llbracket \sigma' \rrbracket : \Omega$
6. If  $\Delta; \Gamma \vdash e : \sigma$  then  $\llbracket \Delta; \Gamma \rrbracket \vdash \llbracket e \rrbracket : \llbracket \sigma \rrbracket$ .

Proof is by induction on the appropriate derivation.

**Theorem 2 (Dynamic Correctness).** If  $\emptyset \vdash e : \sigma$  and  $e \mapsto e'$  then  $\llbracket e \rrbracket \mapsto^* \llbracket e' \rrbracket$ .

Proof is by induction on  $\emptyset \vdash e : \sigma$ .

## 4 Discussion

Despite the simplicity and elegance of this encoding, it falls short for two reasons, which we discuss in this section.

### 4.1 Extension to Primitive Recursion

At the term level we could extend the previous definition of *typerec* to a primitive recursive version *typerec<sup>pr</sup>* by providing terms of type  $R(\alpha)$  and  $R(\beta)$  to  $e_{\rightarrow}$ . In that case,  $\llbracket R(\tau) \rrbracket$  must be a recursive definition:

$$\begin{aligned} \forall \gamma: \llbracket \Omega \rightarrow \Omega \rrbracket. \llbracket T(\gamma \text{ int}) \rrbracket \\ \rightarrow \llbracket \forall \alpha: \Omega. \forall \beta: \Omega. R(\alpha) \rightarrow R(\beta) \rightarrow T(\gamma \alpha) \rightarrow T(\gamma \beta) \rightarrow T(\gamma(\alpha \rightarrow \beta)) \rrbracket \\ \rightarrow \llbracket T(\gamma \tau) \rrbracket \end{aligned}$$

We have defined  $\llbracket R(\tau) \rrbracket$  in terms of  $\llbracket R(\alpha) \rrbracket$  and  $\llbracket R(\beta) \rrbracket$ . We might expect that a realistic term language include parameterized recursive types. In that case, the definition of *typerec<sup>pr</sup>* is no more difficult than that of *typerec*; just supply the extra arguments to the arrow branch. In other words,

$$\begin{aligned} \llbracket R_{\rightarrow} \rrbracket = \Lambda \alpha: \llbracket \Omega \rrbracket. \Lambda \beta: \llbracket \Omega \rrbracket. \lambda x_1: \llbracket R(\alpha) \rrbracket. \lambda x_2: \llbracket R(\beta) \rrbracket. \\ \Lambda \gamma: \llbracket \Omega \rightarrow \Omega \rrbracket. \lambda i. \lambda a. \\ a[\alpha][\beta] \ x_1 \ x_2 \ (x_1[\gamma]ia)(x_2[\gamma]ia) \end{aligned}$$

However, we cannot add recursive kinds to implement primitive recursion at the type constructor level without losing decidable type checking. Even without resorting to recursive types, there is a well known technique for encoding

primitive recursion in terms of iteration, by pairing the argument with the result in the iteration.<sup>2</sup> Unfortunately, this pairing trick only works for closed expressions, and only produces terms that are  $\beta\eta$ -equivalent in the target language. Therefore, at the term level, our strong notion of dynamic correctness does not hold. Using this technique, we must weaken it to:

If  $\emptyset \vdash e : \sigma$  and  $e \mapsto e'$  then  $\llbracket e \rrbracket$  is  $\beta\eta$ -convertible with  $\llbracket e' \rrbracket$ .

At the type-constructor level,  $\beta\eta$ -equivalence is sufficient. However, for type checking, we need the equivalence to extend to constructors with free-variables. The reason that this trick does not work is that  $\lambda U^-$  can encode iteration over datatypes only weakly; there is no induction principle for this encoding provable in  $\lambda U^-$ . Therefore, we cannot derive a proof of equality in the equational theory of the target language that relies on induction. This weakness has been encountered before. In fact, it is conjectured that it is impossible to encode primitive recursion in System F using  $\beta\eta$ -equality [22]. A stronger equational theory for  $\lambda U^-$ , perhaps one incorporating a parametricity principle [19], might solve this problem. However, a simpler way to support primitive recursion would be to include an operator for primitive recursion directly in the language [13,18,3,4].

## 4.2 Impredicativity and Non-termination

Another issue with this encoding is that the target language must have impredicative polymorphism at the type and kind level. In practice, this property is acceptable in the target language. Although, impredicativity at the kind level destroys strong-normalization [2],<sup>3</sup> intensional polymorphism was designed for typed-compilation of Turing-complete language [9], and impredicativity at the type level is vital for such transformations as typed closure conversion. Furthermore, Trifonov *et al.* show that impredicative kind polymorphism allows the analysis of quantified types [23]. Allowing such impredicativity in the source language does not prevent this encoding; we can similarly encode the type-erasure version of their language [21].

However, the source language of this paper,  $\lambda_R$ , is predicative and strongly-normalizing, and the fact that this encoding destroys these properties is unsatisfactory. It seems reasonable, then, to look at methods of encoding iteration within predicative languages [16,7]. In adding iteration to the kind level, *strict* positivity (the recursively bound variable may not appear to the left of an arrow) may be required [3], to prevent the definition of an equivalent paradox.

## 5 Related Work and Conclusions

Böhm and Berarducci [1] showed how to to encode any covariant datatype in the polymorphic lambda calculus. A variant of this idea, called dictionary passing, was used to implement ad-hoc polymorphism in the language Haskell [17]

<sup>2</sup> See the tutorials in Meertens [11] and Mitchell [15] Section 9.3

<sup>3</sup> Coquand [2] originally derived a looping term by formalizing a paradox along the lines of Reynolds' theorem [20], forming an isomorphism between a set and its double power set. Hurkens [10] simplified this argument and developed a shorter looping term, using a related paradox.

through type classes [24]. In Standard ML [14], Yang [25] similarly used it to encode type-specialized functions (such as type-directed partial evaluation [6]). Because core ML does not support higher-order polymorphism, he presented his encoding within the ML module system.

At the type constructor level, Crary and Weirich [4] encoded the *Typerec* construct with a language supporting product, sum and inductive kinds. Their aim was to support type analysis in type-preserving compilation. Because various intermediate languages do not share the same type system, they needed some way to express the analysis of source-level types within the target language.

In this paper we demonstrate that all of these encodings are related, and have the implementation of iteration at their core. While intensional type analysis seems to require highly specialized operators, here we observe that it is no more complicated to include than iteration over inductive datatypes. Though we have implemented such iteration via the standard encoding into the polymorphic lambda calculus, other constructs supporting iteration suffice. In fact, alternative operations for iteration may be necessary in situations where impredicative polymorphism is not desirable.

**Acknowledgments.** Thanks to Robert Harper, Bratin Saha, Karl Crary and Greg Morrisett for much helpful discussion.

## References

1. C. Böhm and A. Berarducci. Automatic synthesis of typed  $\lambda$ -programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.
2. Thierry Coquand. A new paradox in type theory. In Dag Prawitz, Brian Skyrms, and Dag Westerståhl, editors, *Logic, methodology and philosophy of science IX : proceedings of the Ninth International Congress of Logic, Methodology, and Philosophy of Science, Uppsala, Sweden, August 7-14, 1991*, Amsterdam, 1994. Elsevier.
3. Thierry Coquand and Christin Paulin. Inductively defined types. In P. Martin-Löf and G. Mints, editors, *COLOG-88 International Conference on Computer Logic*, volume 417 of *Lecture Notes in Computer Science*, pages 50–66, Tallinn, USSR, December 1988. Springer-Verlag.
4. Karl Crary and Stephanie Weirich. Flexible type analysis. In *1999 ACM International Conference on Functional Programming*, pages 233–248, Paris, September 1999.
5. Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. In *1998 ACM International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 301–313, Baltimore, MD, September 1998. Extended Version is Cornell University Computer Science TR98-1721.
6. Olivier Danvy. Type-directed partial evaluation. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, January 1996.
7. Peter Dybjer. Inductive sets and families in Martin-Löf’s type theory and their set-theoretic semantics. In Gerard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 280–306. Prentice Hall, 1991.
8. Jean-Yves Girard. *Interprétation fonctionnelle et élimination des coupures de l’arithmétique d’ordre supérieur*. PhD thesis, Université Paris VII, 1972.
9. 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.

10. A. J. C. Hurkens. A simplification of girard's paradox. In Mariangiola Dezani-Ciancaglini and Gordon Plotkin, editors, *Second International Conference on Typed Lambda Calculi and Applications, TLCA '95*, volume 902 of *Lecture Notes in Computer Science*, Edinburgh, United Kingdom, April 1995. Springer-Verlag.
11. Lambert G. L. T. Meertens. Paramorphisms. *Formal Aspects of Computing*, 4(5):413–424, 1992.
12. E. Meijer, M.M. Fokkinga, and R. Paterson. Functional programming with bananas, lenses, envelopes and barbed wire. In *FPCA91: Functional Programming Languages and Computer Architecture*, volume 523 of *Lecture Notes in Computer Science*, pages 124–144. Springer-Verlag, 1991.
13. Paul Francis Mendler. *Inductive Definition in Type Theory*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, New York, September 1987.
14. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997.
15. John C. Mitchell. *Foundations for Programming Languages*. The MIT Press, 1996.
16. C. Paulin-Mohring. Inductive definitions in the system Coq - rules and properties. In M. Bezem and J.-F. Groote, editors, *Proceedings of the conference Typed Lambda Calculi and Applications*, number 664 in *Lecture Notes in Computer Science*, 1993. LIP research report 92-49.
17. Simon L. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999. Available from <http://www.haskell.org/definition/>.
18. F. Pfenning and C. Paulin-Mohring. Inductively defined types in the Calculus of Constructions. In *Proceedings of Mathematical Foundations of Programming Semantics*, volume 442 of *Lecture Notes in Computer Science*. Springer-Verlag, 1990.
19. Gordon Plotkin and Martín Abadi. A logic for parametric polymorphism. In *International Conference on Typed Lambda Calculi and Applications*, pages 361–375, 1993.
20. John C. Reynolds. Polymorphism is not set-theoretic. In *Proceedings of the International Symposium on Semantics of Data Types*, volume 173 of *Lecture Notes in Computer Science*. Springer-Verlag, 1984.
21. Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis in type erasure semantics. In *Third Workshop on Types in Compilation*, Montreal, September 2000.
22. Zdzisław Splawski and Paweł Urzyczyn. Type fixpoints: Iteration vs. recursion. In *Fourth ACM International Conference on Functional Programming*, pages 102–113, Paris, France, September 1999.
23. Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM International Conference on Functional Programming*, pages 82–93, Montreal, September 2000. Extended version is YALEU/DCS/TR-1194.
24. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989.
25. Zhe Yang. Encoding types in ML-like languages. In *1998 ACM International Conference on Functional Programming*, volume 34 of *ACM SIGPLAN Notices*, pages 289 – 300, Baltimore, MD, September 1998.

## A Operational Semantics

### A.1 $\lambda_R$

$$(\lambda\alpha:x.e)e' \mapsto e[e'/x]$$

$$(\Lambda\alpha:\kappa.e)[c] \mapsto (e[c/\alpha])$$

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2} \quad \frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

$$\overline{\text{typerec}^{pr}[c] \ R_i \ e_i \ e_{\rightarrow} \mapsto e_i}$$

$$\overline{\text{typerec}^{pr}[c] \ (R_{\rightarrow}[\tau_1][\tau_2]e_1 \ e_2) \ e_i \ e_{\rightarrow} \mapsto}$$

$$e_{\rightarrow}[\tau_1][\tau_2] \ e_1 \ e_2$$

$$(\text{typerec}^{pr}[c] \ e_1 \ e_i \ e_{\rightarrow})$$

$$(\text{typerec}^{pr}[c] \ e_2 \ e_i \ e_{\rightarrow})$$

$$\frac{e \mapsto e'}{\text{typerec}^{pr}[c] \ e \ e_i \ e_{\rightarrow} \mapsto}$$

$$\text{typerec}^{pr}[c] \ e' \ e_i \ e_{\rightarrow}$$

### A.2 $\lambda U^-$

$$(\lambda x:c.e)e' \mapsto e[e'/x]$$

$$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}$$

$$(\Lambda\alpha:\kappa.e)[c] \mapsto (e[c/\alpha])$$

$$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$$

## B Static Semantics of $\lambda_R$

### B.1 Constructor Equivalence

$$\boxed{\Delta \vdash c_1 = c_2 : \kappa}$$

$$\frac{\Delta, \alpha : \kappa' \vdash c_1 : \kappa \quad \Delta \vdash c_2 : \kappa' \quad \alpha \notin \text{Dom}(\Delta)}{\Delta \vdash (\lambda \alpha : \kappa'. c_1) c_2 = c_1 [c_2 / \alpha] : \kappa}$$

$$\frac{\Delta \vdash c : \kappa_1 \rightarrow \kappa_2 \quad \alpha \notin \text{Dom}(\Delta)}{\Delta \vdash \lambda \alpha : \kappa_1. c \alpha = c : \kappa_1 \rightarrow \kappa_2}$$

$$\frac{\Delta, \alpha : \kappa \vdash c = c' : \kappa'}{\Delta \vdash \lambda \alpha : \kappa. c = \lambda \alpha : \kappa. c' : \kappa \rightarrow \kappa'}$$

$$\frac{\Delta \vdash c_1 = c'_1 : \kappa' \rightarrow \kappa \quad \Delta \vdash c_2 = c'_2 : \kappa'}{\Delta \vdash c_1 c_2 = c'_1 c'_2 : \kappa}$$

$$\frac{\Delta \vdash c_1 = c'_1 : \kappa' \rightarrow \kappa \quad \Delta \vdash c_2 = c'_2 : \kappa'}{\Delta \vdash c_1 \rightarrow c_2 = c'_1 \rightarrow c'_2 : \Omega}$$

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

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

$$\frac{\Delta \vdash c = c' : \kappa \quad \Delta \vdash c' = c'' : \kappa}{\Delta \vdash c = c'' : \kappa}$$

$$\frac{\Delta \vdash c_i : \kappa}{\Delta \vdash c_{\rightarrow} : \Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa}$$

$$\Delta \vdash \text{Typerec}^{pr}[\kappa](int)(c_i, c_{\rightarrow}) = c_i : \kappa$$

$$\frac{\Delta \vdash c_1 : \Omega \quad \Delta \vdash c_2 : \Omega \quad \Delta \vdash c_i : \kappa}{\Delta \vdash c_{\rightarrow} : \Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa}$$

$$\Delta \vdash \text{Typerec}^{pr}[\kappa](c_1 \rightarrow c_2)(c_i, c_{\rightarrow}) = c_{\rightarrow} c_1 c_2 (\text{Typerec}^{pr}[\kappa]c_1(c_i, c_{\rightarrow})) (\text{Typerec}^{pr}[\kappa]c_2(c_i, c_{\rightarrow})) : \kappa$$

$$\Delta \vdash c = c' : \Omega$$

$$\Delta \vdash c_i = c'_i : \kappa$$

$$\Delta \vdash c_{\rightarrow} = c'_{\rightarrow} : \Omega \rightarrow \Omega \rightarrow \kappa \rightarrow \kappa$$

$$\Delta \vdash \text{Typerec}^{pr}[\kappa]c(c_i, c_{\rightarrow}) = \text{Typerec}^{pr}[\kappa]c'(c'_i, c'_{\rightarrow}) : \kappa$$

### B.2 Type Equivalence

$$\boxed{\Delta \vdash \sigma_1 = \sigma_2}$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa}{\Delta \vdash T(c_1) = T(c_2)}$$

$$\frac{\Delta \vdash c_1 = c_2 : \kappa}{\Delta \vdash R(c_1) = R(c_2)}$$

$$\frac{\Delta \vdash \sigma_1 = \sigma'_1 \quad \Delta \vdash \sigma_2 = \sigma'_2}{\Delta \vdash \sigma_1 \rightarrow \sigma_2 = \sigma'_1 \rightarrow \sigma'_2}$$

$$\frac{}{\Delta \vdash T(int) = int}$$

$$\frac{\Delta \vdash \sigma_1 = T(c_1) \quad \Delta \vdash \sigma_2 = T(c_2)}{\Delta \vdash \sigma_1 \rightarrow \sigma_2 = T(c_1 \rightarrow c_2)}$$

$$\frac{\Delta, \alpha : \kappa \vdash \sigma = \sigma'}{\Delta \vdash \forall \alpha : \kappa. \sigma = \forall \alpha : \kappa. \sigma'}$$

$$\frac{\Delta \vdash \sigma}{\Delta \vdash \sigma = \sigma}$$

$$\frac{\Delta \vdash \sigma' = \sigma}{\Delta \vdash \sigma = \sigma'}$$

$$\frac{\Delta \vdash \sigma = \sigma' \quad \Delta \vdash \sigma' = \sigma''}{\Delta \vdash \sigma = \sigma''}$$

### B.3 Term Formation

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

$$\frac{}{\Delta; \Gamma \vdash i : int}$$

$$\frac{\Gamma(x) = \sigma}{\Delta; \Gamma \vdash x : \sigma}$$

$$\frac{\Delta; \Gamma, x : \sigma_2 \vdash e : \sigma_1 \quad \Delta; \Gamma \vdash \sigma_2 \quad x \notin \text{Dom}(\Gamma)}{\Delta; \Gamma \vdash \lambda x : \sigma_2. e : \sigma_2 \rightarrow \sigma_1}$$

$$\frac{\Delta; \Gamma \vdash e_1 : \sigma_2 \rightarrow \sigma_1 \quad \Delta; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash e_1 e_2 : \sigma_1}$$

$$\frac{\Delta; \Gamma \vdash e : \forall \alpha : \kappa. \sigma \quad \Delta; \Gamma \vdash c : \kappa}{\Delta; \Gamma \vdash e[c] : \sigma[c/\alpha]}$$

$$\frac{\Delta; \Gamma, \alpha : \kappa \vdash e : \sigma \quad x \notin \text{Dom}(\Gamma)}{\Delta; \Gamma \vdash \Lambda \alpha : \kappa. e : \forall \alpha : \kappa. \sigma}$$

$$\frac{\Delta; \Gamma \vdash e : \sigma_2 \quad \Delta; \Gamma \vdash \sigma_1 = \sigma_2}{\Delta; \Gamma \vdash e : \sigma_1}$$

$$\frac{}{\Delta; \Gamma \vdash R_i : R(int)}$$

$$\begin{array}{c}
\overline{\Delta; \Gamma \vdash R_{\rightarrow} : \forall \alpha: \Omega. \forall \beta: \Omega. \\
R(\alpha) \rightarrow R(\beta) \rightarrow R(\alpha \rightarrow \beta)} \\
\Delta; \Gamma \vdash c : \Omega \rightarrow \Omega \\
\Delta; \Gamma \vdash e : R(\tau) \\
\Delta; \Gamma \vdash e_i : T(c(int)) \\
\Delta; \Gamma \vdash e_{\rightarrow} : \forall \alpha: \Omega. \forall \beta: \Omega. R(\alpha) \rightarrow R(\beta) \\
\rightarrow T(c(\alpha) \rightarrow c(\beta) \rightarrow c(\beta \rightarrow \gamma)) \\
\hline
\Delta; \Gamma \vdash \text{typerec}^{pr}[c] e e_i e_{\rightarrow} : T(c\tau)
\end{array}$$

$$\begin{array}{c}
\overline{E; \Delta \vdash c : \forall \chi'. \kappa} \\
E; \Delta \vdash \Lambda \chi. c[\chi] = c : \forall \chi'. \kappa \\
\hline
E, \chi; \Delta \vdash c = c' : \kappa \\
E; \Delta \vdash \Lambda \chi. c = \Lambda \chi. c' : \forall \chi. \kappa \\
\hline
E; \Delta \vdash c = c' : \forall \chi. \kappa \\
E; \Delta \vdash c[\kappa] = c'[\kappa] : \kappa'[\kappa/\chi]
\end{array}$$

## C Static Semantics of $\lambda U^-$

### C.1 Kind Formation

$$\boxed{E \vdash \kappa}$$

$$\begin{array}{c}
\overline{E, \chi \vdash \chi} \\
\overline{E \vdash \Omega} \\
\frac{E \vdash \kappa_1 \quad E \vdash \kappa_2}{E \vdash \kappa_1 \rightarrow \kappa_2} \\
\frac{E, \chi \vdash \kappa}{E \vdash \forall \chi. \kappa}
\end{array}$$

$$\begin{array}{c}
E; \Delta \vdash c_1 = c'_1 : \kappa' \rightarrow \kappa \\
E; \Delta \vdash c_2 = c'_2 : \kappa' \\
\hline
E; \Delta \vdash c_1 \rightarrow c_2 = c'_1 \rightarrow c'_2 : \Omega
\end{array}$$

$$\begin{array}{c}
\frac{E; \Delta, \alpha: \kappa \vdash \sigma = \sigma'}{E; \Delta \vdash \forall \alpha: \kappa. \sigma = \forall \alpha: \kappa. \sigma'} \\
\frac{E; \Delta \vdash c : \kappa}{E; \Delta \vdash c = c : \kappa} \\
\frac{E; \Delta \vdash c' = c : \kappa}{E; \Delta \vdash c = c' : \kappa} \\
\frac{E; \Delta \vdash c = c' : \kappa \quad E; \Delta \vdash c' = c'' : \kappa}{E; \Delta \vdash c = c'' : \kappa}
\end{array}$$

### C.2 Constructor Equivalence

$$\boxed{E; \Delta \vdash c = c' : \kappa}$$

$$\begin{array}{c}
E; \Delta, \alpha: \kappa' \vdash c_1 : \kappa \\
E; \Delta \vdash c_2 : \kappa' \\
\alpha \notin \text{Dom}(\Delta) \\
\hline
E; \Delta \vdash (\lambda \alpha: \kappa'. c_1) c_2 = c_1[c_2/\alpha] : \kappa \\
\hline
\frac{E; \Delta \vdash c : \kappa_1 \rightarrow \kappa_2 \alpha \notin \text{Dom}(\Delta)}{E; \Delta \vdash \lambda \alpha: \kappa_1. c \alpha = c : \kappa_1 \rightarrow \kappa_2} \\
\hline
\frac{E; \Delta, \alpha: \kappa \vdash c = c' : \kappa'}{E; \Delta \vdash \lambda \alpha: \kappa. c = \lambda \alpha: \kappa. c' : \kappa \rightarrow \kappa'} \\
\hline
\frac{E; \Delta \vdash c_1 = c'_1 : \kappa' \rightarrow \kappa \quad E; \Delta \vdash c_2 = c'_2 : \kappa'}{E; \Delta \vdash c_1 c_2 = c'_1 c'_2 : \kappa} \\
\hline
\frac{E, \chi; \Delta \vdash c : \kappa'}{E; \Delta \vdash \Lambda \chi. c[\kappa] = c[\kappa/\chi] : \kappa'[\kappa/\chi]}
\end{array}$$

### C.3 Term Formation

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

$$\begin{array}{c}
\overline{\Delta; \Gamma \vdash i : \text{int}} \\
\frac{\Gamma(x) = \sigma}{\Delta; \Gamma \vdash x : \sigma} \\
\frac{\Delta; \Gamma, x: \sigma_2 \vdash e : \sigma_1 \quad \Delta; \Gamma \vdash \sigma_2 \quad x \notin \text{Dom}(\Gamma)}{\Delta; \Gamma \vdash \lambda x: \sigma_2. e : \sigma_2 \rightarrow \sigma_1} \\
\frac{\Delta; \Gamma \vdash e_1 : \sigma_2 \rightarrow \sigma_1 \quad \Delta; \Gamma \vdash e_2 : \sigma_2}{\Delta; \Gamma \vdash e_1 e_2 : \sigma_1} \\
\frac{\Delta; \Gamma \vdash e : \forall \alpha: \kappa. \sigma \quad \Delta; \Gamma \vdash c : \kappa}{\Delta; \Gamma \vdash e[c] : \sigma[c/\alpha]} \\
\frac{\Delta; \Gamma, \alpha: \kappa \vdash e : \sigma \quad x \notin \text{Dom}(\Gamma)}{\Delta; \Gamma \vdash \Lambda \alpha: \kappa. e : \forall \alpha: \kappa. \sigma} \\
\frac{\Delta; \Gamma \vdash e : \sigma_2 \quad ; \Delta \vdash \sigma_1 = \sigma_2 : \Omega}{\Delta; \Gamma \vdash e : \sigma_1}
\end{array}$$