# Fully Reflexive Intensional Type Analysis in Type Erasure Semantics[*]

Bratin Saha Valery Trifonov Zhong Shao
Department of Computer Science
Yale University
{saha,trifonov,shao}@cs.yale.edu

August 27, 2000

## Abstract

Compilers for polymorphic languages must support runtime type analysis over arbitrary source language types for coding applications like garbage collection, dynamic linking, pickling, etc. On the other hand, compilers are increasingly being geared to generate type-safe object code. Therefore, it is important to support runtime type analysis in a framework that generates type correct object code. In this paper, we show how to integrate runtime type analysis, over arbitrary types in a source language, into a system that can propagate types through all phases of compilation.

**Keywords:** runtime type analysis, type-safe object code

## 1 Introduction

Modern programming paradigms increasingly rely on applications requiring runtime type analysis, like dynamic linking, garbage collection, and pickling. For example, Java adopts dynamic linking and garbage collection as central features. Distributed programming requires that code and data on one machine be pickled for transmission to a different machine. In a polymorphic language, the compiler must rely on runtime type information to implement these applications. Furthermore, these applications may operate on arbitrary runtime values; therefore, the compiler must support the analysis of arbitrary source language types. We term this ability of analyzing arbitrary source language types as fully reflexive type analysis.

On the other hand, one of the primary goals of a modern compiler is to generate certified code. A certifying compiler [11] is appealing for a number of reasons. We no longer need to trust the correctness of the compiler; instead, we can verify the correctness of the generated code. Checking the correctness of a compiler-generated proof (of a program property) is much easier than proving the correctness of the compiler. Moreover, since we can verify code before executing it, we are no longer restricted to executing code generated only by trusted compilers. A necessary step in building a certifying compiler is to have the compiler generate code that can be type-checked before execution. The type system ensures that the code accesses only the provided resources, makes legal function calls, etc.

Therefore, it is important to support runtime type analysis (over arbitrary source language types) in a framework that can generate type correct object code. Crary et al. [3] proposed a framework that can propagate types through all phases of compilation. The main idea is to construct and pass terms representing types, instead of the types themselves, at runtime. This allows the use of pre-existing term operations to deal with runtime type information. Semantically, singleton types are used to connect a type to its representation. From an implementor's point of view, this framework (hereafter referred to as the CWM framework) seems to simplify some phases in a type preserving compiler; most notably, typed closure conversion [9]. However, the framework proposed in [3] supports the analysis of inductively defined types only; specifically, it does not support the analysis of polymorphic or recursive types. This limits the applicability of their system since most type analyzing applications must deal with recursive objects or polymorphic code blocks.

In this paper, we extend the CWM framework and encode a language supporting fully reflexive type analysis into this framework. The language is based on our previous work [13]; accordingly, it introduces polymorphism at the kind level to handle the analysis of quantified types. This requires a significant extension of the CWM framework. Moreover, even with kind polymorphism, recursive types pose a problem. We deal with this by suitably constraining the analysis of recursive types in the source language, and by using an unconventional rule for fold-unfold expressions in the target language.

The rest of the paper is organized as follows. We give an overview of intensional type analysis in Section 2. We present the source language $\lambda_i^{P+}$ in Section 3. Section 4 shows the target language $\lambda_R^P$ that extends the CWM framework. We offer a translation from $\lambda_i^{P+}$ to $\lambda_R^P$ in Section 5.

## 2 Intensional type analysis

Harper and Morrisett [7] proposed intensional type analysis and presented a type-theoretic framework for expressing computations that analyze types at runtime. They introduced two oper-

[*] This research was sponsored in part by the Defense Advanced Research Projects Agency ISO under the title "Scaling Proof-Carrying Code to Production Compilers and Security Policies," ARPA Order No. H559, issued under Contract No. F30602-99-1-0519, and in part by NSF Grants CCR-9633390 and CCR-9901011. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

ators for explicit type analysis: $\mathsf{typecase}$ for the term level and $\mathsf{Typerec}$ for the type level. For example, a polymorphic subscript function for arrays might be written as the following pseudocode:

$$
\begin{array}{lll}
\mathsf{sub} = \Lambda\alpha.\, \mathsf{typecase}\ \alpha\ \mathsf{of} & & \\
 & \mathsf{int} & \Rightarrow \mathsf{intsub} \\
 & \mathsf{real} & \Rightarrow \mathsf{realsub} \\
 & \beta & \Rightarrow \mathsf{boxedsub}\,[\beta]
\end{array}
$$

Here $\mathsf{sub}$ analyzes the type $\alpha$ of the array elements and returns the appropriate subscript function. We assume that arrays of type $\mathsf{int}$ and $\mathsf{real}$ have specialized representations, say $\mathsf{intarray}$ and $\mathsf{realarray}$, and therefore have specialized subscript functions; all other arrays use the default (boxed) representation.

Typing this subscript function is more interesting, because it must have all of the types $\mathsf{intarray} \to \mathsf{int} \to \mathsf{int}$, $\mathsf{realarray} \to \mathsf{int} \to \mathsf{real}$, and $\mathsf{boxedarray}\,(\alpha) \to \mathsf{int} \to \alpha$ for $\alpha$ other than $\mathsf{int}$ and $\mathsf{real}$. To assign a type to the subscript function, we need a construct at the type level that parallels the $\mathsf{typecase}$ analysis at the term level. The subscript operation would then be typed as

$$
\begin{array}{rclll}
 & \mathsf{sub} & : & \forall\alpha.\, \mathsf{Array}\,(\alpha) \to \mathsf{int} \to \alpha & \\
\text{where} & \mathsf{Array} & = & \lambda\alpha.\, \mathsf{Typecase}\ \alpha\ \mathsf{of} & \\
 & & & \quad \mathsf{int} & \Rightarrow \mathsf{intarray} \\
 & & & \quad \mathsf{real} & \Rightarrow \mathsf{realarray} \\
 & & & \quad \beta & \Rightarrow \mathsf{boxedarray}\,\beta
\end{array}
$$

The $\mathsf{Typecase}$ in the above example is a special case of the $\mathsf{Typerec}$ construct in [7], which supports primitive recursion over types.

## 3 The source language $\lambda_i^{P+}$

We define the syntax of the $\lambda_i^{P+}$ calculus in Figures 1 and 2. The static semantics of $\lambda_i^{P+}$ uses the following three environments:

$$
\begin{array}{llcl}
\text{sort environment} & \mathcal{E} & ::= & \varepsilon \mid \mathcal{E}, \chi \\
\text{kind environment} & \Delta & ::= & \varepsilon \mid \Delta, \alpha\,{:}\,\kappa \\
\text{type environment} & \Gamma & ::= & \varepsilon \mid \Gamma, x\,{:}\,\tau
\end{array}
$$

The detailed typing rules are given in Figures 3 and 4 and the important term reduction rules in Figure 5. The language $\lambda_i^{P+}$ extends the language $\lambda_i^{P}$ proposed in [13] with recursive types, and some additional constructs for analyzing recursive types. This section only gives an overview of the language, the reader may refer to [13] for more details.

In the impredicative $F_\omega$ calculus, the polymorphic types $\forall\alpha : \kappa.\, \tau$ can be viewed as generated by an infinite set of type constructors $\forall_\kappa$ of kind $(\kappa \to \Omega) \to \Omega$, one for each kind $\kappa$. The type $\forall\alpha : \kappa.\, \tau$ is then represented as $\forall_\kappa\,(\lambda\alpha : \kappa.\, \tau)$. The kinds of constructors that can generate types of kind $\Omega$ would then be

$$
\begin{array}{lcl}
\mathsf{int} & : & \Omega \\
\twoheadrightarrow & : & \Omega \to \Omega \to \Omega \\
\forall_\Omega & : & (\Omega \to \Omega) \to \Omega \\
\ldots & & \\
\forall_\kappa & : & (\kappa \to \Omega) \to \Omega \\
\ldots & &
\end{array}
$$

We can avoid the infinite number of $\forall_\kappa$ constructors by defining a single constructor $\boldsymbol{\forall}$ of polymorphic kind $\forall\chi.\, (\chi \to \Omega) \to \Omega$ and then instantiating it to a specific kind before forming polymorphic types. More importantly, this technique also removes the negative occurrence of $\Omega$ from the kind of the argument of the constructor $\forall_\Omega$. Hence in our new $\lambda_i^{P+}$ calculus (see Figures 1 and 2), we extend $F_\omega$ with variable and polymorphic kinds ($\chi$ and $\forall\chi.\, \kappa$) and add a type constant $\boldsymbol{\forall}$ of kind $\forall\chi.\, (\chi \to \Omega) \to \Omega$ to the type language. The polymorphic type $\forall\alpha : \kappa.\, \tau$ is now represented as $\boldsymbol{\forall}\,[\kappa]\,(\lambda\alpha\,{:}\,\kappa.\, \tau)$.

$$
\begin{array}{llcl}
(\mathit{kinds}) & \kappa & ::= & \Omega \mid \kappa \to \kappa' \mid \chi \mid \forall\chi.\, \kappa \\
 & & & \\
(\mathit{types}) & \tau & ::= & \mathsf{int} \mid \twoheadrightarrow \mid \boldsymbol{\forall} \mid \boldsymbol{\forall}^{+} \mid \mu \mid \mathsf{Place} \\
 & & \mid & \alpha \mid \Lambda\chi.\, \tau \mid \lambda\alpha\,{:}\,\kappa.\, \tau \mid \tau\,[\kappa] \mid \tau\,\tau' \\
 & & \mid & \mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^{+}}) \\
 & & & \\
(\mathit{values}) & v & ::= & i \mid \Lambda^{+}\chi.\, v \mid \Lambda\alpha\,{:}\,\kappa.\, v \mid \lambda x\,{:}\,\tau.\, e \mid \mathsf{fix}\, x\,{:}\,\tau.\, v \\
 & & \mid & \mathsf{fold}[\tau]\ v \\
 & & & \\
(\mathit{terms}) & e & ::= & v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
 & & \mid & \mathsf{fold}[\tau]\ e \mid \mathsf{unfold}[\tau]\ e \\
 & & \mid & \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu})
\end{array}
$$

Figure 1: Syntax of the $\lambda_i^{P+}$ language

$$
\begin{array}{rcl}
\tau \to \tau' & \equiv & ((\twoheadrightarrow)\,\tau)\,\tau' \\
\forall\alpha\,{:}\,\kappa.\, \tau & \equiv & (\boldsymbol{\forall}\,[\kappa])\,(\lambda\alpha\,{:}\,\kappa.\, \tau) \\
\forall^{+}\chi.\, \tau & \equiv & \boldsymbol{\forall}^{+}(\Lambda\chi.\, \tau)
\end{array}
$$

Figure 2: Syntactic sugar for $\lambda_i^{P+}$ types

While analyzing a polymorphic type, $\boldsymbol{\forall}\,[\kappa]\,\tau$, the kind $\kappa$ must be held abstract to ensure termination of the analysis [13]. Therefore, the $\mathsf{Typerec}$ operator needs a kind abstraction $\Lambda\chi.\, \tau$ in the branch corresponding to the $\boldsymbol{\forall}$ constructor. We provide kind abstraction and kind application $\tau\,[\kappa]$ at the type level. The formation rules for these constructs, excerpted from Figure 3, are

$$
\frac{\mathcal{E} \vdash \Delta \quad \mathcal{E}, \chi; \Delta \vdash \tau : \kappa}{\mathcal{E}; \Delta \vdash \Lambda\chi.\, \tau : \forall\chi.\, \kappa}
\qquad
\frac{\mathcal{E}; \Delta \vdash \tau : \forall\chi.\, \kappa \qquad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash \tau\,[\kappa'] : \kappa\{\kappa'/\chi\}}
$$

Similarly, at the term level, the $\mathsf{typecase}$ operator must analyze polymorphic types where the quantified type variable may be of an arbitrary kind. To avoid the necessity of analyzing kinds, the $\mathsf{typecase}$ must bind a variable to the kind of the quantified type variable. Therefore, we introduce kind abstraction $\Lambda^{+}\chi.\, v$ and kind application $e\,[\kappa]^{+}$ at the term level. To assign types to these new constructs at the term level, we need a type level construct $\forall^{+}\chi.\, \tau$ that binds the kind variable $\chi$ in the type $\tau$. The formation rules are shown below.

$$
\frac{\mathcal{E}, \chi; \Delta; \Gamma \vdash v : \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda^{+}\chi.\, v : \forall^{+}\chi.\, \tau}
\qquad
\frac{\mathcal{E}; \Delta; \Gamma \vdash e : \forall^{+}\chi.\, \tau \quad \mathcal{E} \vdash \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\,[\kappa]^{+} : \tau\{\kappa/\chi\}}
$$

Furthermore, since our goal is fully reflexive type analysis, we need to analyze kind-polymorphic types as well. As with poly-

$$\boxed{\text{Kind formation} \qquad \mathcal{E} \vdash \kappa}$$

$$\mathcal{E} \vdash \Omega \qquad \frac{\chi \in \mathcal{E}}{\mathcal{E} \vdash \chi} \qquad \frac{\mathcal{E} \vdash \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E} \vdash \kappa \rightarrow \kappa'} \qquad \frac{\mathcal{E}, \chi \vdash \kappa}{\mathcal{E} \vdash \forall \chi.\, \kappa}$$

$$\boxed{\text{Kind environment formation} \qquad \mathcal{E} \vdash \Delta}$$

$$\mathcal{E} \vdash \varepsilon \qquad \frac{\mathcal{E} \vdash \Delta \quad \mathcal{E} \vdash \kappa}{\mathcal{E} \vdash \Delta, \alpha : \kappa}$$

$$\boxed{\text{Type formation} \qquad \mathcal{E}; \Delta \vdash \tau \;:\; \kappa}$$

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{ll} \mathcal{E}; \Delta \vdash \mathsf{int} & : \Omega \\ \mathcal{E}; \Delta \vdash (\twoheadrightarrow) & : \Omega \rightarrow \Omega \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \mathsf{\forall} & : \forall \chi.\, (\chi \rightarrow \Omega) \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \mathsf{\forall}^{+} & : (\forall \chi.\, \Omega) \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \mathsf{\mu} & : (\Omega \rightarrow \Omega) \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \mathsf{Place} & : \Omega \rightarrow \Omega \end{array}}$$

$$\frac{\mathcal{E} \vdash \Delta \quad \alpha : \kappa \text{ in } \Delta}{\mathcal{E}; \Delta \vdash \alpha \;:\; \kappa}$$

$$\frac{\mathcal{E} \vdash \Delta \quad \mathcal{E}, \chi; \Delta \vdash \tau \;:\; \kappa}{\mathcal{E}; \Delta \vdash \Lambda \chi.\, \tau \;:\; \forall \chi.\, \kappa} \qquad \frac{\mathcal{E}; \Delta \vdash \tau \;:\; \forall \chi.\, \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E}; \Delta \vdash \tau\, [\kappa'] \;:\; \kappa\{\kappa'/\chi\}}$$

$$\frac{\mathcal{E}; \Delta, \alpha : \kappa \vdash \tau \;:\; \kappa'}{\mathcal{E}; \Delta \vdash \lambda \alpha : \kappa.\, \tau \;:\; \kappa \rightarrow \kappa'}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau \;:\; \kappa' \rightarrow \kappa \quad \mathcal{E}; \Delta \vdash \tau' \;:\; \kappa'}{\mathcal{E}; \Delta \vdash \tau\, \tau' \;:\; \kappa}$$

$$\frac{\begin{array}{ll} \mathcal{E}; \Delta \vdash \tau & : \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\mathsf{int}} & : \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\rightarrow} & : \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\forall} & : \forall \chi.\, (\chi \rightarrow \Omega) \rightarrow (\chi \rightarrow \Omega) \rightarrow \Omega \\ \mathcal{E}; \Delta \vdash \tau_{\forall^{+}} & : (\forall \chi.\, \Omega) \rightarrow (\forall \chi.\, \Omega) \rightarrow \Omega \end{array}}{\mathcal{E}; \Delta \vdash \mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^{+}}) \;:\; \Omega}$$

$$\boxed{\text{Type environment formation} \qquad \mathcal{E}; \Delta \vdash \Gamma}$$

$$\frac{\mathcal{E} \vdash \Delta}{\mathcal{E}; \Delta \vdash \varepsilon} \qquad \frac{\mathcal{E}; \Delta \vdash \Gamma \quad \mathcal{E}; \Delta \vdash \tau \;:\; \Omega}{\mathcal{E}; \Delta \vdash \Gamma, x : \tau}$$

$$\boxed{\text{Term formation} \qquad \mathcal{E}; \Delta; \Gamma \vdash e \;:\; \tau}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \;:\; \tau \quad \mathcal{E}; \Delta \vdash \tau \leadsto \tau' \;:\; \Omega}{\mathcal{E}; \Delta; \Gamma \vdash e \;:\; \tau'}$$

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

$$\frac{\mathcal{E}; \Delta \vdash \Gamma \quad x : \tau \text{ in } \Gamma}{\mathcal{E}; \Delta; \Gamma \vdash x \;:\; \tau} \qquad \frac{\mathcal{E}, \chi; \Delta; \Gamma \vdash v \;:\; \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda^{+} \chi.\, v \;:\; \forall^{+} \chi.\, \tau}$$

$$\frac{\mathcal{E}; \Delta, \alpha : \kappa; \Gamma \vdash v \;:\; \tau}{\mathcal{E}; \Delta; \Gamma \vdash \Lambda \alpha : \kappa.\, v \;:\; \forall \alpha : \kappa.\, \tau} \qquad \frac{\mathcal{E}; \Delta; \Gamma, x : \tau \vdash e \;:\; \tau'}{\mathcal{E}; \Delta; \Gamma \vdash \lambda x : \tau.\, e \;:\; \tau \rightarrow \tau'}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \;:\; \mathsf{\forall}^{+} \tau \quad \mathcal{E} \vdash \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\, [\kappa]^{+} \;:\; \tau\, [\kappa]}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \;:\; \mathsf{\forall}\, [\kappa]\, \tau \quad \mathcal{E}; \Delta \vdash \tau' \;:\; \kappa}{\mathcal{E}; \Delta; \Gamma \vdash e\, [\tau'] \;:\; \tau\, \tau'}$$

$$\frac{\mathcal{E}; \Delta; \Gamma \vdash e \;:\; \tau' \rightarrow \tau \quad \mathcal{E}; \Delta; \Gamma \vdash e' \;:\; \tau'}{\mathcal{E}; \Delta; \Gamma \vdash e\, e' \;:\; \tau}$$

$$\frac{\begin{array}{c} \mathcal{E}; \Delta; \Gamma, x : \tau \vdash v \;:\; \tau \\ \tau = \forall^{+} \chi_1 \ldots \chi_n.\, \forall \alpha_1 : \kappa_1 \ldots \alpha_m : \kappa_m.\, \tau_1 \rightarrow \tau_2 \\ n \geq 0, m \geq 0 \end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{fix}\, x : \tau.\, v \;:\; \tau}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau \;:\; \Omega \rightarrow \Omega \quad \mathcal{E}; \Delta; \Gamma \vdash e \;:\; \tau\, (\mathsf{\mu} \tau)}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{fold}[\tau]\ e \;:\; \mathsf{\mu} \tau}$$

$$\frac{\mathcal{E}; \Delta \vdash \tau \;:\; \Omega \rightarrow \Omega \quad \mathcal{E}; \Delta; \Gamma \vdash e \;:\; \mathsf{\mu} \tau}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{unfold}[\tau]\ e \;:\; \tau\, (\mathsf{\mu} \tau)}$$

$$\frac{\begin{array}{ll} \mathcal{E}; \Delta \vdash \tau \;:\; \Omega \rightarrow \Omega & \\ \mathcal{E}; \Delta \vdash \tau' \;:\; \Omega & \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\mathsf{int}} & : \tau\, \mathsf{int} \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\rightarrow} & : \forall \alpha : \Omega.\, \forall \alpha' : \Omega.\, \tau\, (\alpha \rightarrow \alpha') \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall} & : \forall^{+} \chi.\, \forall \alpha : \chi \rightarrow \Omega.\, \tau\, (\mathsf{\forall}\, [\chi]\, \alpha) \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\forall^{+}} & : \forall \alpha : (\forall \chi.\, \Omega).\, \tau\, (\mathsf{\forall}^{+} \alpha) \\ \mathcal{E}; \Delta; \Gamma \vdash e_{\mu} & : \forall \alpha : \Omega \rightarrow \Omega.\, \tau\, (\mathsf{\mu}\, \alpha) \end{array}}{\mathcal{E}; \Delta; \Gamma \vdash \mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) \;:\; \tau\, \tau'}$$

Figure 3: Formation rules of $\lambda_i^{P+}$

$$\boxed{\text{Type reduction} \qquad \mathcal{E};\Delta \vdash \tau_1 \rightsquigarrow \tau_2 \,:\, \kappa}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \tau \,:\, \kappa \quad \mathcal{E};\Delta \vdash \tau' \,:\, \kappa'}{\mathcal{E};\Delta \vdash (\lambda\alpha{:}\kappa'.\,\tau)\,\tau' \rightsquigarrow \tau\{\tau'/\alpha\} \,:\, \kappa}$$

$$\frac{\mathcal{E},\chi;\Delta \vdash \tau \,:\, \kappa \quad \mathcal{E} \vdash \kappa'}{\mathcal{E};\Delta \vdash (\Lambda\chi.\,\tau)\,[\kappa'] \rightsquigarrow \tau\{\kappa'/\chi\} \,:\, \kappa\{\kappa'/\chi\}}$$

$$\frac{\mathcal{E};\Delta \vdash \tau \,:\, \kappa \to \kappa' \quad \alpha \notin \mathit{ftv}(\tau)}{\mathcal{E};\Delta \vdash \lambda\alpha{:}\kappa.\,\tau\,\alpha \rightsquigarrow \tau \,:\, \kappa \to \kappa'}$$

$$\frac{\mathcal{E};\Delta \vdash \tau \,:\, \forall\chi'.\,\kappa \quad \chi \notin \mathit{fkv}(\tau)}{\mathcal{E};\Delta \vdash \Lambda\chi.\,\tau\,[\chi] \rightsquigarrow \tau \,:\, \forall\chi'.\,\kappa}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \,:\, \Omega}{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ \mathsf{int}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_{\mathsf{int}} \,:\, \Omega}$$

$$\frac{\begin{array}{c}\mathcal{E};\Delta \vdash \mathsf{Typerec}\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_1' \,:\, \Omega \\ \mathcal{E};\Delta \vdash \mathsf{Typerec}\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_2' \,:\, \Omega\end{array}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Typerec}\ ((\twoheadrightarrow)\,\tau_1\,\tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \qquad \rightsquigarrow \tau_{\to}\,\tau_1\,\tau_2\,\tau_1'\,\tau_2' \,:\, \Omega\end{array}}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\kappa' \vdash \mathsf{Typerec}\ (\tau\,\alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau' \,:\, \Omega}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Typerec}\ (\forall\,[\kappa']\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \qquad \rightsquigarrow \tau_{\forall}\,[\kappa']\,\tau\,(\lambda\alpha{:}\kappa'.\,\tau') \,:\, \Omega\end{array}}$$

$$\frac{\mathcal{E},\chi;\Delta \vdash \mathsf{Typerec}\ (\tau\,[\chi])\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau' \,:\, \Omega}{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ (\forall^{+}\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau_{\forall^+}\,\tau\,(\Lambda\chi.\,\tau') \,:\, \Omega}$$

$$\frac{\mathcal{E};\Delta,\alpha{:}\Omega \vdash \mathsf{Typerec}\ (\tau\,(\mathsf{Place}\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau' \,:\, \Omega}{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ (\mu\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \mu\,(\lambda\alpha{:}\Omega.\,\tau') \,:\, \Omega}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ (\mathsf{Place}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \,:\, \Omega}{\mathcal{E};\Delta \vdash \mathsf{Typerec}\ (\mathsf{Place}\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \tau \,:\, \Omega}$$

Figure 4: Selected $\lambda_i^{P+}$ type reduction rules

morphic types, we can represent the type $\forall^{+}\chi.\,\tau$ as the application of a type constructor $\forall^{+}$ of kind $(\forall\chi.\,\Omega) \to \Omega$ to the type $\Lambda\chi.\,\tau$.

The formation rules in Figure 3 require that the environments are well formed. Moreover, all types and kinds are also well formed. Thus, in the type formation rule $\mathcal{E};\Delta \vdash \tau \,:\, \kappa$, we have that $\mathcal{E} \vdash \Delta$ and $\mathcal{E} \vdash \kappa$. In the term formation rule $\mathcal{E};\Delta;\Gamma \vdash e \,:\, \tau$, we have that $\mathcal{E} \vdash \Delta$ and $\mathcal{E};\Delta \vdash \Gamma$ and $\mathcal{E};\Delta \vdash \tau \,:\, \Omega$.

The Typerec operator is used for type analysis at the type level. In fact, it allows primitive recursion at the type level. It operates on types of kind $\Omega$ and returns a type of kind $\Omega$ (Figure 4). Depending on the head constructor of the type being analyzed, Typerec chooses one of the branches. At the int type, it returns the $\tau_{\mathsf{int}}$ branch. At the function type $\tau \to \tau'$, it applies the $\tau_{\to}$ branch to the components $\tau$ and $\tau'$, and to the result of recursively processing $\tau$ and $\tau'$.

$$\begin{array}{l}\mathsf{Typerec}\ (\tau \to \tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \qquad \tau_{\to}\,\tau\,\tau'\ (\mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+})) \\ \qquad\qquad (\mathsf{Typerec}\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

When analyzing a polymorphic type, the reduction rule is

$$\begin{array}{l}\mathsf{Typerec}\ (\forall\alpha{:}\kappa'.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \tau_{\forall}\,[\kappa']\,(\lambda\alpha{:}\kappa'.\,\tau)\,(\lambda\alpha{:}\kappa'.\,\mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

Since $\tau_{\forall}$ must be parametric in the kind $\kappa'$ (to ensure termination, there are no facilities for kind analysis in the language [13]), it can only apply its second and third arguments to locally introduced type variables of kind $\kappa'$. We believe this restriction, which is crucial for preserving strong normalization of the type language, is quite reasonable in practice. For instance $\tau_{\forall}$ can yield a quantified type based on the result of the analysis.

The reduction rule for analyzing a kind-polymorphic type is

$$\begin{array}{l}\mathsf{Typerec}\ (\forall^{+}\chi.\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \tau_{\forall^+}\,(\Lambda\chi.\,\tau)\,(\Lambda\chi.\,\mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

The $\forall^{+}$-branch of Typerec gets as arguments the body of the quantified type and a kind function encapsulating the result of the analysis on the body of the quantified type.

Recursive types are formed using the $\mu$ constructor of kind $(\Omega \to \Omega) \to \Omega$. For the analysis of recursive types we introduce a unary constructor Place of kind $\Omega \to \Omega$, which is not intended for use by the programmer. There are no term level constructs to create an object whose type contains this constructor. The introduction of the Place constructor is based on the work of Fegaras and Sheard [6].

Recursive types are handled in the manner suggested by the language $\lambda_i^Q$ in [13]. When a recursive type is analyzed by a Typerec, the result is always a recursive type. The Typerec analyzes the body of the type with the $\mu$-bound variable protected under the Place constructor. Since Place is the right inverse of Typerec (Figure 4), the analysis terminates when it reaches a type variable.

$$\begin{array}{l}\mathsf{Typerec}\ (\mu\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \rightsquigarrow \\ \mu\,(\lambda\alpha{:}\Omega.\ \mathsf{Typerec}\ (\tau\,(\mathsf{Place}\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}))\end{array}$$

Since the argument of the $\mu$ constructor has a negative occurrence of the kind $\Omega$, this case must be handled specially. Therefore, the Typerec does not act as an iterator for the $\mu$ constructor. Instead, it directly analyzes the body of the recursive type. In essence, we have made the $\mu$ constructor transparent to the analysis. Operationally, the number of nested $\mu$ constructors strictly decreases at every reduction of the Typerec, ensuring termination after a finite number of steps.

$$
\begin{array}{ll}
\mathsf{typecase}[\tau]\ \mathsf{int}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) & \rightsquigarrow e_{\mathsf{int}} \\
\mathsf{typecase}[\tau]\ (\tau_1 \rightarrow \tau_2)\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) & \rightsquigarrow e_{\rightarrow}\ [\tau_1]\ [\tau_2] \\
\mathsf{typecase}[\tau]\ (\forall\,[\kappa]\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) & \rightsquigarrow e_{\forall}\ [\kappa]^{+}\ [\tau'] \\
\mathsf{typecase}[\tau]\ (\forall^{+}\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) & \rightsquigarrow e_{\forall^{+}}\ [\tau'] \\
\mathsf{typecase}[\tau]\ (\mu\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) & \rightsquigarrow e_{\mu}\ [\tau'] \\
\mathsf{typecase}[\tau]\ (\mathsf{Place}\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) \rightsquigarrow & \\
\qquad\qquad \mathsf{typecase}[\tau]\ (\mathsf{Place}\,\tau')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu}) &
\end{array}
$$

Figure 5: Selected term reduction rules of $\lambda_i^{P+}$

The term expressions are mostly standard. We use explicit fold-unfold operators to implement the isomorphism between a recursive type and its unfolding. Type analysis at the term level is performed using the typecase operator. Since the term level includes a fixed-point operator, typecase is not iterative; it inspects a given type $\tau'$ and passes its components to the corresponding branch. The reduction rules for typecase are in Figure 5.

Existential types can be handled similar to polymorphic types. We define a type constructor $\exists$ of kind $\forall\chi.\,(\chi \rightarrow \Omega) \rightarrow \Omega$. The existential type $\exists\alpha:\kappa.\,\tau$ is then equivalent to $\exists\,[\kappa]\,(\lambda\alpha:\kappa.\,\tau)$. The Typerec and the typecase are augmented with a $\tau_{\exists}$ and a $e_{\exists}$ branch respectively. The reduction rules are exactly analogous to the polymorphic case, except that the Typerec now chooses the $\tau_{\exists}$ branch, and the typecase chooses the $e_{\exists}$ branch.

To illustrate the type level analysis we will use the Typerec operator to define the class of types admitting equality comparisons. We will extend the example in [7] to handle quantified types. We define a type operator $\mathsf{Eq} : \Omega \rightarrow \Omega$ which maps function and polymorphic types to the type Void. (Here $\mathsf{Void} \equiv \forall\alpha:\Omega.\,\alpha$ is a type with no values). To make the example more realistic, we extend the language with a product type constructor ($\divideontimes$) of the same kind as ($\twoheadrightarrow$). The type analysis constructs operate on the $\divideontimes$ constructor in a manner similar to the $\twoheadrightarrow$ constructor.

For ease of presentation we use ML-style pattern matching syntax to define a type involving Typerec: Instead of

$$
\begin{array}{ll}
\multicolumn{2}{l}{\mathsf{t} = \lambda\alpha:\Omega.\ \mathsf{Typerec}\ \alpha\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^{+}})} \\
\text{where} & \tau_{\rightarrow} = \lambda\alpha_1:\Omega.\ \lambda\alpha_2:\Omega.\ \lambda\alpha_1':\kappa.\ \lambda\alpha_2':\kappa.\ \tau_{\rightarrow}' \\
 & \tau_{\forall} = \Lambda\chi.\ \lambda\alpha:\chi \rightarrow \Omega.\ \lambda\alpha':\chi \rightarrow \kappa.\ \tau_{\forall}' \\
 & \tau_{\forall^{+}} = \lambda\alpha:(\forall\chi.\,\Omega).\ \lambda\alpha':(\forall\chi.\,\kappa).\ \tau_{\forall^{+}}'
\end{array}
$$

we write

$$
\begin{array}{lcl}
\mathsf{t}\,(\mathsf{int}) & = & \tau_{\mathsf{int}} \\
\mathsf{t}\,(\alpha_1 \rightarrow \alpha_2) & = & \tau_{\rightarrow}'\{\mathsf{t}\,(\alpha_1), \mathsf{t}\,(\alpha_2)/\alpha_1', \alpha_2'\} \\
\mathsf{t}\,(\forall\,[\chi]\,\alpha) & = & \tau_{\forall}'\{\lambda\alpha_1:\chi.\ \mathsf{t}\,(\alpha\,\alpha_1)/\alpha'\} \\
\mathsf{t}\,(\forall^{+}\alpha) & = & \tau_{\forall^{+}}'\{\Lambda\chi.\ \mathsf{t}\,(\alpha\,[\chi])/\alpha'\}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{fix}\ \mathsf{toString}:\forall\alpha:\Omega.\ \alpha \rightarrow \mathsf{string}. \\
\quad = \Lambda\alpha:\Omega. \\
\qquad \mathsf{typecase}[\lambda\gamma:\Omega.\ \gamma \rightarrow \mathsf{string}]\ \alpha\ \mathsf{of} \\
\qquad\quad \begin{array}{ll}
\mathsf{int} & \Rightarrow \mathsf{intToString} \\
\mathsf{string} & \Rightarrow \lambda x:\mathsf{string}.\ x \\
\divideontimes & \Rightarrow \Lambda\beta_1:\Omega.\ \Lambda\beta_2:\Omega.\ \lambda x:\beta_1 \times \beta_2. \\
 & \qquad \mathsf{toString}\,[\beta_1]\,(\mathsf{x.1})\ \hat{}\ \mathsf{toString}\,[\beta_2]\,(\mathsf{x.2}) \\
\twoheadrightarrow & \Rightarrow \Lambda\beta_1:\Omega.\ \Lambda\beta_2:\Omega.\ \lambda x:\beta_1 \rightarrow \beta_2.\ \text{``}function''\\
\forall & \Rightarrow \Lambda^{+}\chi.\ \Lambda\beta:\chi \rightarrow \Omega.\ \lambda x:\forall\,[\chi]\,\beta.\ \text{``}polymorphic'' \\
\forall^{+} & \Rightarrow \Lambda\beta:\forall\chi.\ \Omega.\ \lambda x:\forall^{+}\beta.\ \text{``}kind\ polymorphic'' \\
\mu & \Rightarrow \Lambda\beta:\Omega \rightarrow \Omega.\ \lambda\mathsf{x}:\mu\,\beta. \\
 & \qquad \mathsf{toString}\,[\beta\,(\mu\,\beta)]\,(\mathsf{unfold}[\beta]\ x)
\end{array}
\end{array}
$$

Figure 6: The function toString

In this syntax the Eq type operator is defined as:

$$
\begin{array}{lcl}
\mathsf{Eq}\,(\mathsf{int}) & = & \mathsf{int} \\
\mathsf{Eq}\,(\alpha_1 \times \alpha_2) & = & \mathsf{Eq}\,(\alpha_1) \times \mathsf{Eq}\,(\alpha_2) \\
\mathsf{Eq}\,(\alpha_1 \rightarrow \alpha_2) & = & \mathsf{Void} \\
\mathsf{Eq}\,(\forall\,[\chi]\,\alpha) & = & \mathsf{Void} \\
\mathsf{Eq}\,(\forall^{+}\alpha) & = & \mathsf{Void} \\
\mathsf{Eq}\,(\mu\,\alpha) & = & \mu\,(\lambda\alpha_1:\Omega.\ \mathsf{Eq}\,(\alpha\,(\mathsf{Place}\,\alpha_1)))
\end{array}
$$

Note that $\mathsf{Eq}\,((\mathsf{int} \rightarrow \mathsf{int}) \times (\mathsf{int} \rightarrow \mathsf{int}))$ reduces to $(\mathsf{Void} \times \mathsf{Void})$; a more complicated definition is necessary to map this type to Void.

As an example of the term level analysis in $\lambda_i^{P+}$, consider the function toString shown in Figure 6. This function uses the type of a value to produce its string representation. (We add the base type string to the language). We assume a primitive function intToString that converts an integer to its string representation, and use ˆ to denote string concatenation.

The language $\lambda_i^{P+}$ has the following properties. The proofs are similar to the proofs for the language $\lambda_i^{P}$ in [13].

**Proposition 3.1 (Strong Normalization)** *Reduction of well formed types is strongly normalizing.*

**Proposition 3.2 (Confluence)** *Reduction of well formed types is confluent.*

**Proposition 3.3 (Type Safety)** *If* $\vdash e:\tau$*, then either* $e$ *is a value, or there exists a term* $e'$ *such that* $e \rightsquigarrow e'$ *and* $\vdash e':\tau$.

### 3.1 Type analysis in $\lambda_i^{P+}$

In our previous work [13], we proposed the language $\lambda_i^{Q}$ which supports the analysis of recursive types without any restrictions. However, the resulting language gets complex and the translation into a CWM framework is not clear. Therefore, type analysis in $\lambda_i^{P+}$ is restricted in two ways. First, the Typerec operator must return a type of kind $\Omega$. Second, the result of analyzing a recursive type is always a recursive type. We believe that these restrictions do not reduce significantly the usefulness of the language in practice.

The main purpose of Typerec is to provide types to typecase terms; every branch of the Typerec types the corresponding branch of the typecase. Since the type of a term is always of kind $\Omega$, the result of the Typerec must also be of kind $\Omega$. Thus, in practice, a Typerec will be employed to form types of kind $\Omega$.

In some cases a Typerec is used to enforce typing constraints—for example, in the case of polymorphic equality above, a Typerec was used to express the constraint that the set of equality types does not include function or polymorphic types. In these cases the Typerec merely verifies that an input type is well formed, while preserving its structure. This means that the Typerec will map a recursive type into a recursive type.

Other applications of type analysis also follow this pattern. Consider a copying collector [14]. The copy function would use a Typerec to express that data from a particular region has been copied into a different region. Since the structure of the data remains the same after being copied, a recursive type would still be mapped into a recursive type. The same holds true while flattening tuples. Flattening involves traversing the input type tree, and converting every tuple into the corresponding flattened type; therefore, the structure of the input type is preserved.

### 3.2 Limitations of type analysis in $\lambda_i^{P+}$

The approach outlined in this section allows the analysis of recursive types within the term language and the type language, but imposes severe limitations on combining these analyses. One can write a polymorphic printer that analyses types at runtime, or one can write a type operator, like Eq, to enforce invariants at the type level. However, it is not possible to write a polymorphic equality function that analyzes types at runtime and has the type $\forall\alpha:\Omega.\,\mathsf{Eq}\,\alpha \rightarrow \mathsf{Eq}\,\alpha \rightarrow \mathsf{bool}$. The reason is that when the recursive type $\mathsf{Eq}\,(\mu\tau)$ is unfolded, the result is $\mathsf{Eq}\,(\tau\,(\mathsf{Place}\,(\mu\tau)))$. The equality function must now analyze the type $\tau\,(\mathsf{Place}\,(\mu\tau))$, which requires it to analyze a Place type. However, the corresponding branch of the typecase can only contain a divergent term. The root of the problem lies in defining Place as a constructor for kind $\Omega$. The solution might require an automatic unfolding of a recursive type when it is being analyzed at the term level; the resulting language exceeds the scope of the current paper. Removing recursive types from the language again allows fully general type analysis.

## 4 The target language $\lambda_R^P$

Figure 7 shows the syntax of the $\lambda_R^P$ language. To make the presentation simpler, we will describe many of the features in the context of the translation from $\lambda_i^{P+}$.

### 4.1 The analyzable components in $\lambda_R^P$

In $\lambda_R^P$, the type calculus is split into types and tags: While types classify terms, tags are used for analysis. We extend the kind calculus to distinguish between the two. The kind $\Omega$ includes the set of types, while the kind $\mathsf{T}$ includes the set of tags. For every constructor that generates a type of kind $\Omega$, we have a corresponding constructor that generates a tag of kind $\mathsf{T}$; for example, int corresponds to $T_{\mathsf{int}}$ and $\twoheadrightarrow$ corresponds to $T_{\rightarrow}$. The type analysis construct at the type level is Tagrec and operates only on tags.

$$
\begin{array}{llcl}
(\mathit{kinds}) & \kappa & ::= & \Omega \mid \mathsf{T} \mid \kappa \rightarrow \kappa' \mid \chi \mid \forall\chi.\,\kappa \\
\\
(\mathit{types}) & \tau & ::= & \mathsf{int} \mid \twoheadrightarrow \mid \forall \mid \forall^{+} \mid \mu \mid \mathsf{Pl} \mid R \\
 & & \mid & T_{\mathsf{int}} \mid T_{\rightarrow} \mid T_{\forall} \mid T_{\forall^{+}} \mid T_{\mu} \mid T_{pl} \mid T_{R} \\
 & & \mid & \alpha \mid \Lambda\chi.\,\tau \mid \tau\,[\kappa] \mid \lambda\alpha:\kappa.\,\tau \mid \tau\,\tau' \\
 & & \mid & \mathsf{Tagrec}\,\tau\ \mathsf{of}\,(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^{+}};\,\tau_{R}) \\
\\
(\mathit{values}) & v & ::= & i \mid \Lambda^{+}\chi.\,v \mid \Lambda\alpha:\kappa.\,v \mid \lambda x:\tau.\,e \mid \mathsf{fix}\,x:\tau.\,v \\
 & & \mid & \mathsf{fold}[\tau]\,v \\
 & & \mid & \mathsf{R}_{\mathsf{int}} \mid \mathsf{R}_{\rightarrow} \mid \mathsf{R}_{\rightarrow}\,[\tau] \mid \mathsf{R}_{\rightarrow}\,[\tau]\,v \\
 & & \mid & \mathsf{R}_{\rightarrow}\,[\tau]\,v\,[\tau'] \mid \mathsf{R}_{\rightarrow}\,[\tau]\,v\,[\tau']\,v' \\
 & & \mid & \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,[\kappa]^{+} \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau] \mid \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau'] \\
 & & \mid & \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau']\,v \\
 & & \mid & \mathsf{R}_{\forall^{+}} \mid \mathsf{R}_{\forall^{+}}\,[\tau] \mid \mathsf{R}_{\forall^{+}}\,[\tau]\,v \\
 & & \mid & \mathsf{R}_{\mu} \mid \mathsf{R}_{\mu}\,[\tau] \mid \mathsf{R}_{\mu}\,[\tau]\,v \\
 & & \mid & \mathsf{R}_{pl} \mid \mathsf{R}_{pl}\,[\tau] \mid \mathsf{R}_{pl}\,[\tau]\,v \\
 & & \mid & \mathsf{R}_{R} \mid \mathsf{R}_{R}\,[\tau] \mid \mathsf{R}_{R}\,[\tau]\,v \\
\\
(\mathit{terms}) & e & ::= & v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
 & & \mid & \mathsf{fold}[\tau]\,e \mid \mathsf{unfold}[\tau]\,e \\
 & & \mid & \mathsf{repcase}[\tau]\,e\ \mathsf{of}\,(e_{\mathsf{int}};\,e_{\rightarrow};\,e_{\forall};\,e_{\forall^{+}};\,e_{R};\,e_{\mu};\,e_{pl})
\end{array}
$$

Figure 7: Syntax of the $\lambda_R^P$ language

At the term level, we add representations for tags. The term level operator (now called repcase) analyzes these representations. All the primitive tags have corresponding term level representations; for example, $T_{\mathsf{int}}$ is represented by $\mathsf{R}_{\mathsf{int}}$. Given any tag, the corresponding term representation can be constructed inductively.

### 4.2 Typing term representations

The type calculus in $\lambda_R^P$ includes a unary type constructor $R$ of kind $\mathsf{T} \rightarrow \Omega$ to type the term level representations. Given a tag $\tau$ (of kind $\mathsf{T}$), the term representation of $\tau$ has the type $R\,\tau$. For example, $\mathsf{R}_{\mathsf{int}}$ has the type $R\,T_{\mathsf{int}}$. Semantically, $R\,\tau$ is interpreted as a singleton type that is inhabited only by the term representation of $\tau$ [3].

If the tag $\tau$ is of a function kind $\kappa \rightarrow \kappa'$, then the term representation of $\tau$ is a polymorphic function from representations to representations:

$$R_{\kappa\rightarrow\kappa'}\,(\tau) \equiv \forall\beta:\kappa.\,R_{\kappa}\,(\beta) \rightarrow R_{\kappa'}\,(\tau\,\beta)$$

However a problem arises if $\tau$ is of a variable kind $\chi$. The only way of knowing the type of its representation $R_{\chi}$ is to construct it when $\chi$ is instantiated. Hence programs translated into $\lambda_R^P$ must be such that for every kind variable $\chi$ in the program, a corresponding type variable $\alpha_{\chi}$, representing the type of the term representation for a tag of kind $\chi$, is also available.

This is why we need to extend the CWM framework. Their source language does not include kind polymorphism; therefore,

$$
\begin{array}{ll}
|\Omega| = \mathsf{T} & |\kappa \to \kappa'| = |\kappa| \to |\kappa'| \\
|\chi| = \chi & |\forall\chi.\,\kappa| = \forall\chi.\,(\chi \to \Omega) \to |\kappa|
\end{array}
$$

Figure 8: Translation of $\lambda_i^{P+}$ kinds to $\lambda_R^P$ kinds

they can compute the type of all the representations statically. This is also the reason that we need to introduce a new set of primitive type constructors and split the type calculus into types and tags. Consider the $\forall$ and the $\overset{+}{\forall}$ type constructors in $\lambda_i^{P+}$. The $\forall$ constructor binds a kind $\kappa$. When it is translated into $\lambda_R^P$, the translated constructor must also, in addition, bind a type of kind $\kappa \to \Omega$. Therefore, we need a new constructor $T_\forall$. Similarly, the $\overset{+}{\forall}$ constructor binds a type function of kind $\forall\chi.\,\Omega$. When it is translated into $\lambda_R^P$, the translated constructor must bind a type function of kind $|\forall\chi.\,\Omega|$. (See Figure 8.) Therefore, we introduce a new constructor $T_{\overset{+}{\forall}}$. Furthermore, if we only have $\Omega$ as the primitive kind, it will no longer be inductive. (The inductive definition would break for $T_{\overset{+}{\forall}}$). Therefore, we introduce a new kind $\mathsf{T}$ (for tags), and allow analysis only over tags.

This leads us to the kind translation from $\lambda_i^{P+}$ to $\lambda_R^P$ (Figure 8). Since the analyzable component of $\lambda_R^P$ is of kind $\mathsf{T}$, the $\lambda_i^{P+}$ kind $\Omega$ is mapped to $\mathsf{T}$. The polymorphic kind $\forall\chi.\,\kappa$ is translated to $\forall\chi.\,(\chi \to \Omega) \to |\kappa|$. Note that every kind variable $\chi$ must now have a corresponding type variable $\alpha_\chi$. Given a tag of variable kind $\chi$, the type of its term representation is given by $\alpha_\chi$. Since the type of a term is always of kind $\Omega$, the variable $\alpha_\chi$ has the kind $\chi \to \Omega$.

**Lemma 4.1** $|\kappa\{\kappa'/\chi\}| = |\kappa|\{|\kappa'|/\chi\}$

**Proof** By induction over the structure of $\kappa$. □

Figure 9 shows the function $R_\kappa$. Suppose $\tau$ is a $\lambda_i^{P+}$ type of kind $\kappa$ and $|\tau|$ is its translation into $\lambda_R^P$. The function $R_\kappa$ gives the type of the term representation of $|\tau|$. Since this function is used by the translation from $\lambda_i^{P+}$ to $\lambda_R^P$, it is defined by induction on $\lambda_i^{P+}$ kinds.

**Lemma 4.2** $(R_\kappa)\{|\kappa'|, R_{\kappa'}/\chi', \alpha_{\chi'}\} = R_{\kappa\{\kappa'/\chi'\}}$

**Proof** By induction over the structure of $\kappa$. □

The formation rules for tags are displayed in Figure 10. Since the translation maps $\lambda_i^{P+}$ type constructors to these tags, a type constructor of kind $\kappa$ is mapped to a corresponding tag of kind $|\kappa|$. Thus, while the $\forall$ type constructor has the kind $\forall\chi.\,(\chi \to \Omega) \to \Omega$, the $T_\forall$ tag has the kind $\forall\chi.\,(\chi \to \Omega) \to (\chi \to \mathsf{T}) \to \mathsf{T}$.

Figure 10 also shows the type of the term representation of the primitive type constructors. These types agree with the definition of the function $R_\kappa$; for example, the type of $\mathsf{R}_\to$ is $R_{\Omega\to\Omega\to\Omega}\,(T_\to)$. The term formation rules in Figure 10 use a tag interpretation function $\mathsf{F}$ that is explained in Section 4.4.

$$
\frac{\mathcal{E} \vdash \Delta}{\mathcal{E};\Delta \vdash R_\Omega \equiv R \;:\; \mathsf{T} \to \Omega}
\qquad
\frac{\mathcal{E};\Delta \vdash \alpha_\chi \;:\; \chi \to \Omega}{\mathcal{E};\Delta \vdash R_\chi \equiv \alpha_\chi \;:\; \chi \to \Omega}
$$

$$
\frac{\mathcal{E};\Delta \vdash R_\kappa \equiv \tau \;:\; |\kappa| \to \Omega \qquad \mathcal{E};\Delta \vdash R_{\kappa'} \equiv \tau' \;:\; |\kappa'| \to \Omega}{\begin{array}{l}\mathcal{E};\Delta \vdash R_{\kappa\to\kappa'} \equiv \lambda\alpha\!:\!|\kappa \to \kappa'|.\,\forall\beta\!:\!|\kappa|.\,\tau\,\beta \to \tau'\,(\alpha\,\beta)\\ \qquad :\; |\kappa \to \kappa'| \to \Omega\end{array}}
$$

$$
\frac{\mathcal{E},\chi;\Delta,\alpha_\chi\!:\!\chi \to \Omega \vdash R_\kappa \equiv \tau \;:\; |\kappa| \to \Omega}{\begin{array}{l}\mathcal{E};\Delta \vdash R_{\forall\chi.\,\kappa} \equiv \lambda\alpha\!:\!|\forall\chi.\,\kappa|.\,\overset{+}{\forall}\chi.\,\forall\alpha_\chi\!:\!\chi \to \Omega.\,\tau\,(\alpha\,[\chi]\,\alpha_\chi)\\ \qquad :\; |\forall\chi.\,\kappa| \to \Omega\end{array}}
$$

Figure 9: Types of representations at higher kinds

### 4.3 Tag analysis in $\lambda_R^P$

We now consider the tag analysis constructs in more detail. The term level analysis is done by the $\mathsf{repcase}$ construct. Figure 10 and Figure 11 show its static and dynamic semantics respectively. The expression being analyzed must be of type $R\,\tau$; therefore, $\mathsf{repcase}$ always analyzes term representation of tags. Operationally, it examines the representation at the head, selects the corresponding branch, and passes the components of the representation to the selected branch. Thus the rule for analyzing the representation of a polymorphic type is

$$
\begin{array}{l}\mathsf{repcase}[\tau]\ \mathsf{R}_\forall\ [\kappa]^{+}\,[\tau_\kappa]\,[\tau']\,(e')\ \mathsf{of} \\ \quad (e_{\mathsf{int}};\ e_\to;\ e_\forall;\ e_{\overset{+}{\forall}};\ e_R;\ e_\mu;\ e_{pl})\end{array}
\leadsto\ e_\forall\,[\kappa]^{+}\,[\tau_\kappa]\,[\tau']\,(e')
$$

Type level analysis is performed by the $\mathsf{Tagrec}$ construct. The language must be fully reflexive, so $\mathsf{Tagrec}$ includes an additional branch for the new type constructor $T_{_R}$. Since only the kind of $T_\mu$ contains the kind $\mathsf{T}$ in a doubly-negative position (Figure 10), we can define $\mathsf{Tagrec}$ as an iterator over the kind $\mathsf{T}$, and treat $T_\mu$ specially (like the μ constructor in $\lambda_i^{P+}$).

Figure 12 shows the reduction rules for the $\mathsf{Tagrec}$, which are similar to the reduction rules for the source language $\mathsf{Typerec}$: given a tag, it calls itself recursively on the components of the tag and then passes the result of the recursive calls, along with the original components, to the corresponding branch. Thus the reduction rule for the function tag is

$$
\begin{array}{l}
\mathsf{Tagrec}\ (T_\to\ \tau\ \tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\overset{+}{\forall}};\ \tau_{_R}) \leadsto \\
\qquad \tau_\to\ \tau\ \tau'\ (\mathsf{Tagrec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\overset{+}{\forall}};\ \tau_{_R})) \\
\qquad\qquad (\mathsf{Tagrec}\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\overset{+}{\forall}};\ \tau_{_R}))
\end{array}
$$

Similarly, the reduction for the polymorphic tag is

$$
\begin{array}{l}
\mathsf{Tagrec}\ (T_\forall\ [\kappa]\ \tau_\kappa\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\overset{+}{\forall}};\ \tau_{_R}) \leadsto \\
\tau_\forall\ [\kappa]\ \tau_\kappa\ \tau\ (\lambda\alpha\!:\!\kappa.\,\mathsf{Tagrec}\ (\tau\ \alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_\to;\ \tau_\forall;\ \tau_{\overset{+}{\forall}};\ \tau_{_R}))
\end{array}
$$

Recursive tags are handled in a manner similar to recursive types in $\lambda_i^{P+}$. The result is constrained to be a recursive tag. The analysis proceeds directly to the body of the tag function, with the bound variable protected under a $T_{pl}$ tag, which is the right inverse of $\mathsf{Tagrec}$.

The reduction rules also include a rule for the $\mathsf{Pl}$ constructor. The $\mathsf{Pl}$ constructor is used to handle recursive tags in the $\mathsf{F}$ function (Section 4.4). This constructor is again an implementation

$$\boxed{\text{Type formation} \qquad \mathcal{E};\Delta \vdash \tau \,:\, \kappa}$$

$$\frac{\mathcal{E} \vdash \Delta}{\begin{array}{lcl} \mathcal{E};\Delta \vdash R & : & \mathsf{T} \to \Omega \\ \mathcal{E};\Delta \vdash \mathsf{Pl} & : & \Omega \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\mathsf{int}} & : & \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\to} & : & \mathsf{T} \to \mathsf{T} \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\forall} & : & \forall\chi.\,(\chi \to \Omega) \to (\chi \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\forall^+} & : & (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{\mu} & : & (\mathsf{T} \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{pl} & : & \mathsf{T} \to \mathsf{T} \\ \mathcal{E};\Delta \vdash T_{R} & : & \mathsf{T} \to \mathsf{T} \end{array}}$$

$$\frac{\begin{array}{lcl} \mathcal{E};\Delta \vdash \tau & : & \mathsf{T} \\ \mathcal{E};\Delta \vdash \tau_{\mathsf{int}} & : & \mathsf{T} \\ \mathcal{E};\Delta \vdash \tau_{\to} & : & \mathsf{T} \to \mathsf{T} \to \mathsf{T} \to \mathsf{T} \to \mathsf{T} \\ \mathcal{E};\Delta \vdash \tau_{\forall} & : & \forall\chi.\,(\chi \to \Omega) \to (\chi \to \mathsf{T}) \to (\chi \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash \tau_{\forall^+} & : & (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to (\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to \mathsf{T} \\ \mathcal{E};\Delta \vdash \tau_{R} & : & \mathsf{T} \to \mathsf{T} \to \mathsf{T} \end{array}}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \,:\, \mathsf{T}}$$

$$\boxed{\text{Term formation} \qquad \mathcal{E};\Delta;\Gamma \vdash e \,:\, \tau}$$

$$\frac{\mathcal{E};\Delta \vdash \Gamma}{\begin{array}{lcl} \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\mathsf{int}} & : & R\,T_{\mathsf{int}} \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\to} & : & R_{\Omega\to\Omega\to\Omega}\,(T_{\to}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall} & : & R_{\forall\chi.\,(\chi\to\Omega)\to\Omega}\,(T_{\forall}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\forall^+} & : & R_{(\forall\chi.\,\Omega)\to\Omega}\,(T_{\forall^+}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{R} & : & R_{\Omega\to\Omega}\,(T_{R}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{\mu} & : & R_{(\Omega\to\Omega)\to\Omega}\,(T_{\mu}) \\ \mathcal{E};\Delta;\Gamma \vdash \mathsf{R}_{pl} & : & R_{\Omega\to\Omega}\,(T_{pl}) \end{array}}$$

$$\frac{\mathcal{E};\Delta \vdash \tau \,:\, \mathsf{T} \to \mathsf{T} \qquad \mathcal{E};\Delta;\Gamma \vdash e \,:\, \mathsf{F}\,(\tau\,(T_{\mu}\,\tau))}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{fold}[\tau]\ e \,:\, \mathsf{F}\,(T_{\mu}\,\tau)}$$

$$\frac{\mathcal{E};\Delta \vdash \tau \,:\, \mathsf{T} \to \mathsf{T} \qquad \mathcal{E};\Delta;\Gamma \vdash e \,:\, \mathsf{F}\,(T_{\mu}\,\tau)}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{unfold}[\tau]\ e \,:\, \mathsf{F}\,(\tau\,(T_{\mu}\,\tau))}$$

$$\frac{\begin{array}{lcl} \mathcal{E};\Delta \vdash \tau & : & \mathsf{T} \to \Omega \\ \mathcal{E};\Delta;\Gamma \vdash e & : & R\,\tau' \\ \mathcal{E};\Delta;\Gamma \vdash e_{\mathsf{int}} & : & \tau\,T_{\mathsf{int}} \\ \mathcal{E};\Delta;\Gamma \vdash e_{\to} & : & \forall\alpha_1\!:\!\mathsf{T}.\,R\,\alpha_1 \to \forall\alpha_2\!:\!\mathsf{T}.\,R\,\alpha_2 \to \tau\,(T_{\to}\,\alpha_1\,\alpha_2) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall} & : & \forall^{+}\chi.\,\forall\alpha_{\chi}\!:\!\chi \to \Omega. \\ & & \quad \forall\alpha\!:\!\chi \to \mathsf{T}.\,R_{\chi\to\Omega}\,(\alpha) \to \tau\,(T_{\forall}\,[\chi]\,\alpha_{\chi}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\forall^+} & : & \forall\alpha\!:\!\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}.\,R_{\forall\chi.\,\Omega}\,(\alpha) \to \tau\,(T_{\forall^+}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{R} & : & \forall\alpha\!:\!\mathsf{T}.\,R\,\alpha \to \tau\,(T_{R}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{\mu} & : & \forall\alpha\!:\!\mathsf{T} \to \mathsf{T}.\,R_{\Omega\to\Omega}\,(\alpha) \to \tau\,(T_{\mu}\,\alpha) \\ \mathcal{E};\Delta;\Gamma \vdash e_{pl} & : & \forall\alpha\!:\!\mathsf{T}.\,R\,\alpha \to \tau\,(T_{pl}\,\alpha) \end{array}}{\mathcal{E};\Delta;\Gamma \vdash \mathsf{repcase}[\tau]\ e\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \,:\, \tau\,\tau'}$$

Figure 10: Formation rules for the new constructs in $\lambda_R^P$

$$\mathsf{repcase}[\tau]\ \mathsf{R}_{\mathsf{int}}\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto e_{\mathsf{int}}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{\to}\,[\tau_1]\,(e_1)\,[\tau_2]\,(e_2)\ \mathsf{of} \\ \quad (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto e_{\to}\,[\tau_1]\,(e_1)\,[\tau_2]\,(e_2) \end{array}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau_{\kappa}]\,[\tau']\,(e')\ \mathsf{of} \\ \quad (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto e_{\forall}\,[\kappa]^{+}\,[\tau_{\kappa}]\,[\tau']\,(e') \end{array}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{\forall^+}\,[\tau']\,(e')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto \\ \qquad e_{\forall^+}\,[\tau']\,(e') \end{array}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{R}\,[\tau']\,(e')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto \\ \qquad e_{R}\,[\tau']\,(e') \end{array}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{\mu}\,[\tau']\,e'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto \\ \qquad e_{\mu}\,[\tau']\,(e') \end{array}$$

$$\begin{array}{l} \mathsf{repcase}[\tau]\ \mathsf{R}_{pl}\,[\tau']\,(e')\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{R};\ e_{\mu};\ e_{pl}) \leadsto \\ \qquad e_{pl}\,[\tau']\,(e') \end{array}$$

Figure 11: Selected term reduction rules of $\lambda_R^P$

artifact in $\lambda_R^P$ and has no counterpart in the source language. Its reduction rule will never be used in a program translated from $\lambda_i^{P+}$.

### 4.4 The tag interpretation function

Programs in $\lambda_R^P$ pass tags at runtime since only tags can be analyzed. However, abstractions and the fixpoint operator must still carry type information for type checking. Therefore, these annotations must use a function mapping tags to types. Since these annotations are always of kind $\Omega$, this function must map tags of kind $\mathsf{T}$ to types of kind $\Omega$. This implies that we can use an iterator over tags to define the function as follows:

$$\begin{array}{lcl} \mathsf{F}\,(T_{\mathsf{int}}) & = & \mathsf{int} \\ \mathsf{F}\,(T_{\to}\,\alpha_1\,\alpha_2) & = & \mathsf{F}\,(\alpha_1) \to \mathsf{F}\,(\alpha_2) \\ \mathsf{F}\,(T_{\forall}\,[\chi]\,\alpha_{\chi}\,\alpha) & = & \forall\beta\!:\!\chi.\,\alpha_{\chi}\,\beta \to \mathsf{F}\,(\alpha\,\beta) \\ \mathsf{F}\,(T_{\forall^+}\,\alpha) & = & \forall\chi.\,\forall\alpha_{\chi}\!:\!\chi \to \Omega.\,\mathsf{F}\,(\alpha\,[\chi]\,\alpha_{\chi}) \\ \mathsf{F}\,(T_{\mu}\,\alpha) & = & \mu(\lambda\beta\!:\!\Omega.\,\mathsf{F}\,(\alpha\,(\mathsf{Pl}\,\beta))) \\ \mathsf{F}\,(\mathsf{Pl}\,\alpha) & = & \alpha \\ \mathsf{F}\,(T_{R}\,\alpha) & = & \mathsf{int} \\ \mathsf{F}\,(T_{pl}\,\alpha) & = & \mathsf{int} \end{array}$$

The function $\mathsf{F}$ takes a type tree in the $\mathsf{T}$ kind space and converts it into the corresponding tree in the $\Omega$ kind space. Therefore, it converts the tag $T_{\mathsf{int}}$ to the type $\mathsf{int}$. For the other tags, it recursively converts the components into the corresponding types. The branches for the $T_R$ and the $T_{pl}$ tags are bogus but of the correct kind. The language $\lambda_R^P$ is only intended as a target for translation from $\lambda_i^{P+}$—the only interesting programs in $\lambda_R^P$ are the ones translated from $\lambda_i^{P+}$; therefore, the $T_R$ branch of $\mathsf{F}$ will remain unused. Similarly, since the source language hides the Place constructor completely from the programmer, it does not appear in $\lambda_i^{P+}$ programs; hence the $T_{pl}$ branch of $\mathsf{F}$ will also remain unused.

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ T_{\mathsf{int}}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R)\ :\ \mathsf{T}}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ T_{\mathsf{int}}\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau_{\mathsf{int}}\ :\ \mathsf{T}}$$

$$\frac{\begin{array}{c}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ \tau_1\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau_1'\ :\ \mathsf{T}\\ \mathcal{E};\Delta \vdash \mathsf{Tagrec}\ \tau_2\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau_2'\ :\ \mathsf{T}\end{array}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_{\rightarrow}\ \tau_1\ \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad \tau_{\rightarrow}\ \tau_1\ \tau_2\ \tau_1'\ \tau_2'\ :\ \mathsf{T}\end{array}}$$

$$\frac{\begin{array}{l}\mathcal{E};\Delta,\alpha:\kappa' \vdash \mathsf{Tagrec}\ (\tau_2\ \alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad \tau'\ :\ \mathsf{T}\end{array}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_{\forall}\ [\kappa']\ \tau_1\ \tau_2)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad \tau_{\forall}\ [\kappa']\ \tau_1\ \tau_2\ (\lambda\alpha:\kappa'.\,\tau')\ :\ \mathsf{T}\end{array}}$$

$$\frac{\begin{array}{l}\mathcal{E},\chi;\Delta,\alpha_\chi:\chi\rightarrow\Omega \vdash\\ \quad \mathsf{Tagrec}\ (\tau\ [\chi]\ \alpha_\chi)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau'\ :\ \mathsf{T}\end{array}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_{\forall^+}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad \tau_{\forall^+}\ \tau\ (\Lambda\chi.\,\lambda\alpha_\chi:\chi\rightarrow\Omega.\,\tau')\ :\ \mathsf{T}\end{array}}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau'\ :\ \mathsf{T}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_R\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad \tau_R\ \tau\ \tau'\ :\ \mathsf{T}\end{array}}$$

$$\frac{\begin{array}{l}\mathcal{E};\Delta,\alpha:\mathsf{T} \vdash\\ \quad \mathsf{Tagrec}\ (\tau\ (T_{pl}\ \alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau'\ :\ \mathsf{T}\end{array}}{\begin{array}{l}\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_\mu\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow\\ \qquad T_\mu\ (\lambda\alpha:\mathsf{T}.\,\tau')\ :\ \mathsf{T}\end{array}}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_{pl}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R)\ :\ \mathsf{T}}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (T_{pl}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \tau\ :\ \mathsf{T}}$$

$$\frac{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (\mathsf{Pl}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R)\ :\ \mathsf{T}}{\mathcal{E};\Delta \vdash \mathsf{Tagrec}\ (\mathsf{Pl}\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_R) \rightsquigarrow \mathsf{Pl}\ \tau\ :\ \mathsf{T}}$$

Figure 12: Reduction rules for $\lambda_R^P$ Typerec

The type interpretation function has the following properties.

**Lemma 4.3** $(\mathsf{F}\,(\tau))\{\tau'/\alpha\} = \mathsf{F}\,(\tau\{\tau'/\alpha\})$

**Proof** Follows from the fact that none of the branches of $\mathsf{F}$ has free type variables. □

**Lemma 4.4** $(\mathsf{F}\,(\tau))\{\kappa/\chi\} = \mathsf{F}\,(\tau\{\kappa/\chi\})$

**Proof** Follows from the fact that none of the branches of $\mathsf{F}$ has free kind variables. □

The language $\lambda_R^P$ has the following properties.

**Proposition 4.5 (Type Reduction)** *Reduction of well formed types is strongly normalizing and confluent.*

**Proposition 4.6 (Type Safety)** *If* $\vdash e:\tau$*, then either* $e$ *is a value, or there exists a term* $e'$ *such that* $e \rightsquigarrow e'$ *and* $\vdash e':\tau$*.*

$$\begin{array}{ll}
|\alpha| = \alpha & \\
|\mathsf{int}| = T_{\mathsf{int}} & |\Lambda\chi.\,\tau| = \Lambda\chi.\,\lambda\alpha_\chi:\chi\rightarrow\Omega.\,|\tau| \\
|\twoheadrightarrow| = T_{\rightarrow} & |\tau\ [\kappa]| = |\tau|\ [|\kappa|]\ R_\kappa \\
|\forall| = T_{\forall} & |\lambda\alpha:\kappa.\,\tau| = \lambda\alpha:|\kappa|.\,|\tau| \\
|\forall^{+}| = T_{\forall^+} & |\tau\ \tau'| = |\tau|\ |\tau'| \\
|\mu| = T_\mu & |\mathsf{Place}| = T_{pl}
\end{array}$$

$$\begin{array}{l}
|\mathsf{Typerec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+})| = \\
\mathsf{Tagrec}\ |\tau|\ \mathsf{of}\ (|\tau_{\mathsf{int}}|;\ |\tau_{\rightarrow}|;\ |\tau_{\forall}|;\ |\tau_{\forall^+}|;\ \lambda\_:\mathsf{T}.\,\lambda\_:\mathsf{T}.\,|\tau_{\mathsf{int}}|)
\end{array}$$

Figure 13: Translation of $\lambda_i^{P+}$ types to $\lambda_R^P$ tags

$$\begin{array}{rcl}
|i| &=& i \\
|x| &=& x \\
|\Lambda^{+}\chi.\,v| &=& \Lambda^{+}\chi.\,\Lambda\alpha_\chi:\chi\rightarrow\Omega.\,|v| \\
|e\ [\kappa]^{+}| &=& |e|\ [|\kappa|]^{+}\ [R_\kappa] \\
|\Lambda\alpha:\kappa.\,v| &=& \Lambda\alpha:|\kappa|.\,\lambda x_\alpha:R_\kappa\,\alpha.\,|v| \\
|e\ [\tau]| &=& |e|\ [|\tau|]\ \Re(\tau) \\
|\lambda x:\tau.\,e| &=& \lambda x:\mathsf{F}\,|\tau|.\,|e| \\
|e\ e'| &=& |e|\ |e'| \\
|\mathsf{fix}\ x:\tau.\,v| &=& \mathsf{fix}\ x:\mathsf{F}\,|\tau|.\,|v| \\
|\mathsf{fold}[\tau]\ e| &=& \mathsf{fold}[|\tau|]\ |e| \\
|\mathsf{unfold}[\tau]\ e| &=& \mathsf{unfold}[|\tau|]\ |e| \\
\multicolumn{3}{l}{|\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^+};\ e_\mu)|} \\
\multicolumn{3}{l}{\quad = \mathsf{repcase}[\lambda\alpha:\mathsf{T}.\,\mathsf{F}\,(|\tau|\,\alpha)]\ \Re(\tau')\ \mathsf{of}} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_{\mathsf{int}} \Rightarrow |e_{\mathsf{int}}|} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_{\rightarrow} \Rightarrow |e_{\rightarrow}|} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_{\forall} \Rightarrow |e_{\forall}|} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_{\forall^+} \Rightarrow |e_{\forall^+}|} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_R \Rightarrow \Lambda\beta:\mathsf{T}.\,\lambda x:R\,\beta.\,\mathsf{fix}\ x:\mathsf{F}\,(|\tau|\ (T_R\,\beta)).\,x} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_\mu \Rightarrow |e_\mu|} \\
\multicolumn{3}{l}{\qquad\quad \mathsf{R}_{pl} \Rightarrow \Lambda\beta:\mathsf{T}.\,\lambda x:R\,\beta.\,\mathsf{fix}\ x:\mathsf{F}\,(|\tau|\ (T_{pl}\,\beta)).\,x}
\end{array}$$

Figure 14: Translation of $\lambda_i^{P+}$ terms to $\lambda_R^P$ terms

## 5 Translation from $\lambda_i^{P+}$ to $\lambda_R^P$

In this section, we show a translation from $\lambda_i^{P+}$ to $\lambda_R^P$. The languages differ mainly in two ways. First, the type calculus in $\lambda_R^P$ is split into tags and types, with types used solely for type checking and tags used for analysis. Therefore, type passing in $\lambda_i^{P+}$ will get converted into tag passing in $\lambda_R^P$. Second, the typecase operator in $\lambda_i^{P+}$ must be converted into a repcase operating on term representation of tags.

Figure 13 shows the translation of $\lambda_i^{P+}$ types into $\lambda_R^P$ tags. The primitive type constructors get translated into the corresponding tag constructors. The Typerec gets converted into a Tagrec. The translation inserts an arbitrarily chosen well-kinded result into the branch for the $T_R$ tag since the source contains no such branch.

The term translation is shown in Figure 14. The translation

$$
\begin{aligned}
\Re(\mathsf{int}) &= \mathsf{R}_{\mathsf{int}}\\
\Re(\twoheadrightarrow) &= \Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\,\Lambda\beta\!:\!\mathsf{T}.\,\lambda x_\beta\!:\!R\,\beta.\\
&\qquad \mathsf{R}_{\rightarrow}\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta)\\
\Re(\forall) &= \Lambda^{+}\chi.\,\Lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,\Lambda\alpha\!:\!\chi\rightarrow\mathsf{T}.\,\lambda x_\alpha\!:\!R_{\chi\rightarrow\Omega}\,(\alpha).\\
&\qquad \mathsf{R}_{\forall}\,[\chi]^{+}\,[\alpha_\chi]\,[\alpha]\,(x_\alpha)\\
\Re(\forall^{+}) &= \Lambda\alpha\!:\!(\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\mathsf{T}).\,\lambda x_\alpha\!:\!R_{\forall\chi.\,\Omega}\,(\alpha).\\
&\qquad \mathsf{R}_{\forall^{+}}\,[\alpha]\,(x_\alpha)\\
\Re(\mu) &= \Lambda\alpha\!:\!\mathsf{T}\rightarrow\mathsf{T}.\,\lambda\alpha_x\!:\!R_{\Omega\rightarrow\Omega}\,(\alpha).\,\mathsf{R}_\mu\,[\alpha]\,(x_\alpha)\\
\Re(\mathsf{Place}) &= \Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\,\mathsf{R}_{pl}\,[\alpha]\,(x_\alpha)\\
\Re(\alpha) &= x_\alpha\\
\Re(\Lambda\chi.\,\tau) &= \Lambda^{+}\chi.\,\Lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,\Re(\tau)\\
\Re(\tau\,[\kappa]) &= \Re(\tau)\,[|\kappa|]^{+}\,[R_\kappa]\\
\Re(\lambda\alpha\!:\!\kappa.\,\tau) &= \Lambda\alpha\!:\!|\kappa|.\,\lambda x_\alpha\!:\!R_\kappa\,\alpha.\,\Re(\tau)\\
\Re(\tau\,\tau') &= \Re(\tau)\,[|\tau'|]\,(\Re(\tau'))
\end{aligned}
$$

$$
\begin{aligned}
&\Re(\mathsf{Typerec}\,\tau\,\mathsf{of}\,(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^{+}}))\;=\\
&(\mathsf{fix}\,\mathsf{f}\!:\!\forall\alpha\!:\!\mathsf{T}.\,R\,\alpha\rightarrow R\;(\tau^{*}\,\alpha).\\
&\quad \Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\\
&\quad \mathsf{repcase}[\lambda\alpha\!:\!\mathsf{T}.\,R\;(\tau^{*}\,\alpha)]\,x_\alpha\;\mathsf{of}\\
&\qquad \mathsf{R}_{\mathsf{int}}\Rightarrow\Re(\tau_{\mathsf{int}})\\
&\qquad \mathsf{R}_{\rightarrow}\Rightarrow\Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\,\Lambda\beta\!:\!\mathsf{T}.\,\lambda x_\beta\!:\!R\,\beta.\\
&\qquad\qquad \Re(\tau_{\rightarrow})\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta)\\
&\qquad\qquad\quad [\tau^{*}\,\alpha]\,(\mathsf{f}\,[\alpha]\,x_\alpha)\,[\tau^{*}\,\beta]\,(\mathsf{f}\,[\beta]\,x_\beta)\\
&\qquad \mathsf{R}_{\forall}\;\Rightarrow\Lambda^{+}\chi.\,\Lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,\Lambda\alpha\!:\!\chi\rightarrow\mathsf{T}.\,\lambda x_\alpha\!:\!R_{\chi\rightarrow\Omega}\,(\alpha).\\
&\qquad\qquad \Re(\tau_{\forall})\,[\chi]^{+}\,[\alpha_\chi]\,[\alpha]\,(x_\alpha)\,[\lambda\beta\!:\!\chi.\,\tau^{*}\,(\alpha\,\beta)]\\
&\qquad\qquad\quad (\Lambda\beta\!:\!\chi.\,\lambda x_\beta\!:\!\alpha_\chi\,\beta.\,\mathsf{f}\,[\alpha\,\beta]\,(x_\alpha\,[\beta]\,x_\beta))\\
&\qquad \mathsf{R}_{\forall^{+}}\Rightarrow\Lambda\alpha\!:\!(\forall\chi.\,(\chi\rightarrow\Omega)\rightarrow\mathsf{T}).\,\lambda x_\alpha\!:\!R_{\forall\chi.\,\Omega}\,(\alpha).\\
&\qquad\qquad \Re(\tau_{\forall^{+}})\,[\alpha]\,(x_\alpha)\,[\Lambda\chi.\,\lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,\tau^{*}\,(\alpha\,[\chi]\,\alpha_\chi)]\\
&\qquad\qquad (\Lambda^{+}\chi.\,\Lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,\mathsf{f}\,[\alpha\,[\chi]\,\alpha_\chi]\,(x_\alpha\,[\chi]^{+}\,[\alpha_\chi]))\\
&\qquad \mathsf{R}_{R}\;\Rightarrow\Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\,\Re(\tau_{\mathsf{int}})\\
&\qquad \mathsf{R}_{\mu}\;\Rightarrow\Lambda\alpha\!:\!\mathsf{T}\rightarrow\mathsf{T}.\,\lambda x_\alpha\!:\!R_{\Omega\rightarrow\Omega}\,(\alpha).\\
&\qquad\qquad \mathsf{R}_\mu\,[\lambda\beta\!:\!\mathsf{T}.\,\tau^{*}\,(\alpha\,(T_{pl}\,\beta))]\\
&\qquad\qquad (\Lambda\beta\!:\!\mathsf{T}.\,\lambda x_\beta\!:\!R\,\beta.\\
&\qquad\qquad\quad \mathsf{f}\,[\alpha\,(T_{pl}\,\beta)]\,(x_\alpha\,[T_{pl}\,\beta]\,(\mathsf{R}_{pl}\,[\beta]\,(x_\beta))))\\
&\qquad \mathsf{R}_{pl}\;\Rightarrow\Lambda\alpha\!:\!\mathsf{T}.\,\lambda x_\alpha\!:\!R\,\alpha.\,x_\alpha)\\
&[|\tau|]\\
&\Re(\tau)\\
&\text{where}\\
&\quad \tau^{*}=|\lambda\alpha\!:\!\Omega.\,\mathsf{Typerec}\,\alpha\,\mathsf{of}\,(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^{+}})|
\end{aligned}
$$

Figure 15: Representation of $\lambda_i^{P+}$ types as $\lambda_R^{P}$ terms

must maintain two invariants. First, for every accessible kind variable $\chi$, it adds a corresponding type variable $\alpha_\chi$; this variable gives the type of the term representation for a tag of kind $\chi$. At every kind application, the translation uses the function $R_\kappa$ (Figure 9) to compute this type. Thus, the translations of kind abstractions and kind applications are

$$|\Lambda^{+}\chi.\,v| = \Lambda^{+}\chi.\,\Lambda\alpha_\chi\!:\!\chi\rightarrow\Omega.\,|v| \qquad |e\,[\kappa]^{+}| = |e|\,[|\kappa|]^{+}\,[R_\kappa]$$

Second, for every accessible type variable $\alpha$, a term variable $x_\alpha$ is introduced, providing the corresponding term representation of $\alpha$. At every type application, the translation uses the function $\Re(\tau)$ (Figure 15) to construct this representation. Furthermore, type application gets replaced by an application to a tag, and to the term representation of the tag. Thus the translations for type abstractions and type applications are

$$|\Lambda\alpha\!:\!\kappa.\,v| = \Lambda\alpha\!:\!|\kappa|.\,\lambda x_\alpha\!:\!R_\kappa\,\alpha.\,|v| \qquad |e\,[\tau]| = |e|\,[|\tau|]\,\Re(\tau)$$

As pointed out before, the translations of abstraction and the fixpoint operator use the tag interpretation function $\mathsf{F}$ to map tags to types.

We show the term representation of types in Figure 15. The primitive type constructors get translated to the corresponding term representation. The representations of type and kind functions are similar to the term translation of type and kind abstractions. The only involved case is the term representation of a $\mathsf{Typerec}$. Since $\mathsf{Typerec}$ is recursive, we use a combination of a $\mathsf{repcase}$ and a $\mathsf{fix}$. We will illustrate only one case here; the other cases can be reasoned about similarly.

Consider the reduction of $\mathsf{Ty}\,(\tau'\rightarrow\tau'')$, where $\mathsf{Ty}\,\tau$ stands for $\mathsf{Typerec}\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\rightarrow};\,\tau_{\forall};\,\tau_{\forall^{+}})$. This type reduces to $\tau_{\rightarrow}\,\tau'\,\tau''\,(\mathsf{Ty}\,(\tau'))\,(\mathsf{Ty}\,(\tau''))$ . Therefore, in the translation, the term representation of $\tau_{\rightarrow}$ must be applied to the term representations of $\tau'$, $\tau''$, and the result of the recursive calls to the $\mathsf{Typerec}$. The representations of $\tau'$ and $\tau''$ are bound to the variables $x_\alpha$ and $x_\beta$; by assumption the representations for the results of the recursive calls are obtained from the recursive calls to the function $\mathsf{f}$.

In the following propositions the original $\lambda_i^{P+}$ kind environment $\Delta$ is extended with a kind environment $\Delta(\mathcal{E})$ which binds a type variable $\alpha_\chi$ of kind $\chi\rightarrow\Omega$ for each $\chi\in\mathcal{E}$. Similarly the term-level translations extend the type environment $\Gamma$ with $\Gamma(\Delta)$, binding a variable $x_\alpha$ of type $R_\kappa\,\alpha$ for each type variable $\alpha$ bound in $\Delta$ with kind $\kappa$.

**Proposition 5.1** *If* $\mathcal{E};\Delta\;\vdash\;\tau\;:\;\kappa$ *holds in* $\lambda_i^{P+}$*, then* $|\mathcal{E}|;|\Delta|,\,\Delta(\mathcal{E})\vdash|\tau|\;:\;|\kappa|$ *holds in* $\lambda_R^{P}$*.*

**Proof** Follows directly by induction over the structure of $\tau$. □

**Proposition 5.2** *If* $\mathcal{E};\Delta\vdash\tau\;:\;\kappa$ *and* $\mathcal{E};\Delta\vdash\Gamma$ *hold in* $\lambda_i^{P+}$*, then* $|\mathcal{E}|;|\Delta|,\,\Delta(\mathcal{E});|\Gamma|,\,\Gamma(\Delta)\vdash\Re(\tau)\;:\;R_\kappa\,|\tau|$ *holds in* $\lambda_R^{P}$*.*

**Proof** By induction over the structure of $\tau$. The only interesting case is that of a kind application which uses Lemma 4.2. □

**Proposition 5.3** *If* $\mathcal{E};\Delta;\Gamma\;\vdash\;e\;:\;\tau$ *holds in* $\lambda_i^{P+}$*, then* $|\mathcal{E}|;|\Delta|,\,\Delta(\mathcal{E});|\Gamma|,\,\Gamma(\Delta)\vdash|e|\;:\;\mathsf{F}\,|\tau|$ *holds in* $\lambda_R^{P}$*.*

**Proof** This is proved by induction over the structure of $e$, using Lemmas 4.3 and 4.4. □

## 6 The untyped language

This section shows that in $\lambda_R^{P}$ types are not necessary for computation. Figure 16 shows an untyped language $\lambda_R^{P\circ}$. We show a translation from $\lambda_R^{P}$ to $\lambda_R^{P\circ}$ in Figure 17. The expression 1 is the integer constant one.

$$
\begin{array}{lrcl}
(\mathit{values}) & v & ::= & i \mid \lambda x.e \mid \mathsf{fix}\,x.v \mid \mathsf{fold}\;v \\
 & & \mid & \mathsf{R_{int}} \mid \mathsf{R}_{\rightarrow} \mid \mathsf{R}_{\rightarrow}\,1 \mid \mathsf{R}_{\rightarrow}\,1\,v \\
 & & \mid & \mathsf{R}_{\rightarrow}\,1\,v\,1 \mid \mathsf{R}_{\rightarrow}\,1\,v\,1\,v' \\
 & & \mid & \mathsf{R}_{\forall} \mid \mathsf{R}_{\forall}\,1 \mid \mathsf{R}_{\forall}\,1\,1 \mid \mathsf{R}_{\forall}\,1\,1\,1 \\
 & & \mid & \mathsf{R}_{\forall}\,1\,1\,1\,v \\
 & & \mid & \mathsf{R}_{\forall^+} \mid \mathsf{R}_{\forall^+}\,1 \mid \mathsf{R}_{\forall^+}\,1\,v \\
 & & \mid & \mathsf{R}_{\mu} \mid \mathsf{R}_{\mu}\,1 \mid \mathsf{R}_{\mu}\,1\,v \\
 & & \mid & \mathsf{R}_{pl} \mid \mathsf{R}_{pl}\,1 \mid \mathsf{R}_{pl}\,1\,v \\
 & & \mid & \mathsf{R}_{R} \mid \mathsf{R}_{R}\,1 \mid \mathsf{R}_{R}\,1\,v \\
 & & & \\
(\mathit{terms}) & e & ::= & v \mid x \mid e\,e' \mid \mathsf{fold}\;e \mid \mathsf{unfold}\;e \\
 & & \mid & \mathsf{repcase}\;e\;\mathsf{of}\;(e_{\mathsf{int}};\;e_{\rightarrow};\;e_{\forall};\;e_{\forall^+};\;e_R;\;e_\mu;\;e_{pl})
\end{array}
$$

Figure 16: Syntax of the untyped language $\lambda_R^{P\circ}$

$$
\begin{aligned}
i^\circ &= i \\
(\Lambda^{+}\chi.\,v)^\circ &= \lambda\_.v^\circ \\
(\Lambda\alpha:\kappa.\,v)^\circ &= \lambda\_.v^\circ \\
(\lambda x:\tau.\,e)^\circ &= \lambda x.e^\circ \\
(\mathsf{fix}\,x:\tau.\,v)^\circ &= \mathsf{fix}\,x.v^\circ \\
(\mathsf{fold}[\tau]\;e)^\circ &= \mathsf{fold}\;e^\circ \\
(\mathsf{unfold}[\tau]\;e)^\circ &= \mathsf{unfold}\;e^\circ \\
(e\,[\kappa]^{+})^\circ &= e^\circ\,1 \\
(e\,[\tau])^\circ &= e^\circ\,1 \\
(e\,e_1)^\circ &= e^\circ\,e_1{}^\circ \\
\mathsf{R_{int}}^\circ &= \mathsf{R_{int}} \\
\mathsf{R}_{\rightarrow}{}^\circ &= \mathsf{R}_{\rightarrow} \\
(\mathsf{R}_{\rightarrow}\,[\tau])^\circ &= \mathsf{R}_{\rightarrow}\,1 \\
(\mathsf{R}_{\rightarrow}\,[\tau]\,e)^\circ &= \mathsf{R}_{\rightarrow}\,1\,e^\circ \\
(\mathsf{R}_{\rightarrow}\,[\tau]\,e\,[\tau'])^\circ &= \mathsf{R}_{\rightarrow}\,1\,e^\circ\,1 \\
(\mathsf{R}_{\rightarrow}\,[\tau]\,e\,[\tau']\,e_1)^\circ &= \\
&\quad \mathsf{R}_{\rightarrow}\,1\,e^\circ\,1\,e_1{}^\circ
\end{aligned}
\qquad
\begin{aligned}
\mathsf{R}_{\forall}{}^\circ &= \mathsf{R}_{\forall} \\
(\mathsf{R}_{\forall}\,[\kappa]^{+})^\circ &= \mathsf{R}_{\forall}\,1 \\
(\mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau])^\circ &= \mathsf{R}_{\forall}\,1\,1 \\
(\mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau'])^\circ &= \mathsf{R}_{\forall}\,1\,1\,1 \\
(\mathsf{R}_{\forall}\,[\kappa]^{+}\,[\tau]\,[\tau']\,e)^\circ &= \mathsf{R}_{\forall}\,1\,1\,1\,e^\circ \\
\mathsf{R}_{\forall^+}{}^\circ &= \mathsf{R}_{\forall^+} \\
(\mathsf{R}_{\forall^+}\,[\tau])^\circ &= \mathsf{R}_{\forall^+}\,1 \\
(\mathsf{R}_{\forall^+}\,[\tau]\,e)^\circ &= \mathsf{R}_{\forall^+}\,1\,e^\circ \\
\mathsf{R}_{\mu}{}^\circ &= \mathsf{R}_{\mu} \\
(\mathsf{R}_{\mu}\,[\tau])^\circ &= \mathsf{R}_{\mu}\,1 \\
(\mathsf{R}_{\mu}\,[\tau]\,e)^\circ &= \mathsf{R}_{\mu}\,1\,e^\circ \\
\mathsf{R}_{pl}{}^\circ &= \mathsf{R}_{pl} \\
(\mathsf{R}_{pl}\,[\tau])^\circ &= \mathsf{R}_{pl}\,1 \\
(\mathsf{R}_{pl}\,[\tau]\,e)^\circ &= \mathsf{R}_{pl}\,1\,e^\circ \\
\mathsf{R}_{R}{}^\circ &= \mathsf{R}_{R} \\
(\mathsf{R}_{R}\,[\tau])^\circ &= \mathsf{R}_{R}\,1 \\
(\mathsf{R}_{R}\,[\tau]\,e)^\circ &= \mathsf{R}_{R}\,1\,e^\circ
\end{aligned}
$$

$$
\begin{aligned}
(\mathsf{repcase}[\tau]\;e\;\mathsf{of}\;(e_{\mathsf{int}};\;e_{\rightarrow};\;e_{\forall};\;e_{\forall^+};\;e_R;\;e_\mu;\;e_{pl}))^\circ &= \\
\mathsf{repcase}\;e^\circ\;\mathsf{of}\;(e_{\mathsf{int}}{}^\circ;\;e_{\rightarrow}{}^\circ;\;e_{\forall}{}^\circ;\;e_{\forall^+}{}^\circ;\;e_R{}^\circ;\;e_\mu{}^\circ;\;e_{pl}{}^\circ)&
\end{aligned}
$$

Figure 17: Translation of $\lambda_R^{P}$ to $\lambda_R^{P\circ}$

The translation replaces type and kind applications (abstractions) by a dummy application (abstraction), instead of erasing them. In the typed language, a type or a kind can be applied to a fixpoint. This results in an unfolding of the fixpoint. Therefore, the translation inserts dummy applications to preserve this unfolding.

The untyped language has the following property which shows that term reduction in $\lambda_R^{P\circ}$ parallels term reduction in $\lambda_R^{P}$.

**Proposition 6.1** *If* $e \leadsto^* e_1$, *then* $e^\circ \leadsto^* e_1{}^\circ$.

## 7 Related work and conclusions

Our work closely follows the framework proposed in Crary et al. [3]. However, as we pointed before, they consider a language that analyzes inductively defined types only. Extending the analysis to arbitrary types makes the translation much more complicated. The splitting of the type calculus into types and tags, and defining an interpretation function to map between the two, is related to the ideas proposed by Crary and Weirich for the language LX [2], which provides a rich kind calculus and a construct for primitive recursion over types. This allows the user to define new kinds and recursive operations over types of these kinds.

This framework also resembles the dictionary passing style in Haskell [12]. The term representation of a type may be viewed as the dictionary corresponding to the type. However, the authors consider dictionary passing in an untyped calculus; moreover, they do not consider the intensional analysis of types. Dubois et al. [4] also pass explicit type representations in their extensional polymorphism scheme. However, they do not provide a mechanism for connecting a type to its representation. Minamide's [8] type-lifting procedure is also related to our work. His procedure maintains interrelated constraints between type parameters; however, his language does not support intensional type analysis.

Duggan [5] proposes another framework for intensional type analysis. His system allows for the analysis of types at the term level only. It adds a facility for defining type classes and allows type analysis to be restricted to members of such classes. Yang [15] presents some approaches to enable type-safe programming of type-indexed values in ML which is similar to term level analysis of types. Aspinall [1] studied a typed $\lambda$-calculus with subtypes and singleton types.

Necula [11] proposed the idea of a certifying compiler and showed the construction of a certifying compiler for a type-safe subset of C. Morrisett et al. [10] showed that a fully type preserving compiler generating type safe assembly code is a practical basis for a certifying compiler.

We have presented a framework that supports the analysis of arbitrary source language types, but does not require explicit type passing. Instead, term level representations of types are passed at runtime. This allows the use of term level constructs to handle type information at runtime.

## References

[1] D. Aspinall. Subtyping with singleton types. In *Proc. 1994 CSL*. Springer Lecture Notes in Computer Science, 1995.

[2] K. Crary and S. Weirich. Flexible type analysis. In *Proc. 1999 ACM SIGPLAN International Conference on Functional Programming*, pages 233–248. ACM Press, Sept. 1999.

[3] K. Crary, S. Weirich, and G. Morrisett. Intensional polymorphism in type-erasure semantics. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 301–312. ACM Press, Sept. 1998.

[4] C. Dubois, F. Rouaix, and P. Weis. Extensional polymorphism. In *Proc. 22nd Annual ACM Symp. on Principles of Programming Languages*, pages 118–129. ACM Press, 1995.

[5] D. Duggan. A type-based semantics for user-defined marshalling in polymorphic languages. In X. Leroy and A. Ohori, editors, *Proc.*

*1998 International Workshop on Types in Compilation*, volume 1473 of *LNCS*, pages 273–298, Kyoto, Japan, Mar. 1998. Springer-Verlag.

[6] L. Fegaras and T. Sheard. Revisiting catamorphism over datatypes with embedded functions. In *23rd Annual ACM Symp. on Principles of Programming Languages*, pages 284–294. ACM Press, Jan. 1996.

[7] R. Harper and G. Morrisett. Compiling polymorphism using intensional type analysis. In *Proc. 22nd Annual ACM Symp. on Principles of Programming Languages*, pages 130–141. ACM Press, Jan. 1995.

[8] Y. Minamide. Full lifting of type parameters. Technical report, RIMS, Kyoto University, 1997.

[9] Y. Minamide, G. Morrisett, and R. Harper. Typed closure conversion. In *Proc. 23rd Annual ACM Symp. on Principles of Programming Languages*, pages 271–283. ACM Press, 1996.

[10] G. Morrisett, D. Walker, K. Crary, and N. Glew. From System F to typed assembly language. In *Proc. 25th Annual ACM Symp. on Principles of Programming Languages*, pages 85–97. ACM Press, Jan. 1998.

[11] G. C. Necula. *Compiling with Proofs*. PhD thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, Sept. 1998.

[12] J. Peterson and M. Jones. Implementing type classes. In *Proc. ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 227–236. ACM Press, June 1993.

[13] V. Trifonov, B. Saha, and Z. Shao. Fully reflexive intensional type analysis. In *Proc. 2000 ACM SIGPLAN International Conference on Functional Programming*. ACM Press, 2000.

[14] D. Wang and A. Appel. Safe garbage collection = regions + intensional type analysis. Technical report, Dept. of Computer Science, Princeton University, July 1999.

[15] Z. Yang. Encoding types in ML-like languages. In *Proc. 1998 ACM SIGPLAN International Conference on Functional Programming*, pages 289–300. ACM Press, 1998.

## A The language $\lambda_i^{P+}$

In this section, we compare $\lambda_i^{P+}$ with the languages proposed in [13], specially for those unfamiliar with the previous work.

The language $\lambda_i^{P+}$ is a simplification of the language $\lambda_i^Q$ proposed in [13] which allows unrestricted type analysis. In $\lambda_i^Q$, the result of the Typerec is not restricted to be of kind $\Omega$; moreover, while analyzing a recursive type, the Typerec is not required to return another recursive type. The kind and type calculus of $\lambda_i^Q$ is shown below, while the term calculus is identical.

$$\begin{array}{lrcl}
(kinds) & \kappa & ::= & \chi \mid \natural\kappa \mid \kappa \to \kappa' \mid \forall\chi.\,\kappa \\[1ex]
(types) & \tau & ::= & \alpha \mid \mathsf{int} \mid \overset{\circ}{\twoheadrightarrow} \mid \mathring{\forall} \mid \mathring{\forall}^{+} \mid \mathring{\mu} \mid \mathsf{Place} \\
& & \mid & \lambda\alpha\!:\!\kappa.\,\tau \mid \tau\,\tau' \mid \Lambda\chi.\,\tau \mid \tau\,[\kappa] \\
& & \mid & \mathsf{Typerec}\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\to};\,\tau_{\forall};\,\tau_{\forall^+};\,\tau_{\mu})
\end{array}$$

To allow general type analysis, $\lambda_i^Q$ uses parameterized kinds $\natural\kappa$. The kind $\Omega$ is then modeled as $\forall\chi.\,\natural\chi$. The Typerec analyzes types of kind $\natural\kappa$, and returns a type of kind $\kappa$. (Refer to [13] for details). The presence of parameterized kinds complicates the system and makes the translation into a type erasure semantics unclear.

On the other hand, the language $\lambda_i^P$ [13] shown below has a translation into a type erasure semantics. This language supports fully general analysis of polymorphic types.

$$\begin{array}{lrcl}
(kinds) & \kappa & ::= & \Omega \mid \kappa \to \kappa' \mid \chi \mid \forall\chi.\,\kappa \\[1ex]
(types) & \tau & ::= & \mathsf{int} \mid \twoheadrightarrow \mid \forall \mid \forall^{+} \\
& & \mid & \alpha \mid \Lambda\chi.\,\tau \mid \lambda\alpha\!:\!\kappa.\,\tau \mid \tau\,[\kappa] \mid \tau\,\tau' \\
& & \mid & \mathsf{Typerec}\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\,\tau_{\to};\,\tau_{\forall};\,\tau_{\forall^+}) \\[1ex]
(values) & v & ::= & i \mid \Lambda^{+}\chi.\,e \mid \Lambda\alpha\!:\!\kappa.\,e \mid \lambda x\!:\!\tau.\,e \mid \mathsf{fix}\,x\!:\!\tau.\,v \\[1ex]
(terms) & e & ::= & v \mid x \mid e\,[\kappa]^{+} \mid e\,[\tau] \mid e\,e' \\
& & \mid & \mathsf{typecase}[\tau]\;\tau'\;\mathsf{of}\;(e_{\mathsf{int}};\,e_{\to};\,e_{\forall};\,e_{\forall^+})
\end{array}$$

Therefore, to support fully reflexive type analysis in a type erasure semantics, we must base our language on $\lambda_i^P$. Accordingly, to arrive at $\lambda_i^{P+}$, we augment the type calculus of $\lambda_i^P$ with recursive types and recursive type analysis. Correspondingly, we extend the term calculus of $\lambda_i^P$ with fold and unfold operators, and add a branch for recursive types to the typecase operator. This, in turn, requires that we restrict type analysis suitably to ensure a sound and decidable type system.