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

June 1, 2000

Abstract

Compilers for polymorphic languages must support runtime type analysis over arbitrary 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, 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 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 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 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.

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 operators for explicit type analysis: typecase for the term level and Typerec for the type level. For example, a polymorphic subscript function for arrays might be written as the following pseudo-

*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.

code:

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

Here sub analyzes the type α of the array elements and returns the appropriate subscript function. We assume that arrays of type int and 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 α other than int and 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

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

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

The rest of the paper is organized as follows. We present the source language λ_i^{P+} in Section 3. Section 4 shows the target language λ_R^P that extends the CWM framework. We offer a translation from λ_i^{P+} to λ_R^P in Section 5.

3 The source language λ_i^{P+}

We define the syntax of the λ_i^{P+} calculus in Figure 1. The static semantics of λ_i^{P+} uses the following three environments:

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

The detailed typing rules are given in Figures 3 and 4 and the important term reduction rules in Figure 5. The language λ_i^{P+} extends the language λ_i^P proposed in [13] with recursive types and supports the analysis of arbitrary source language types. This section only gives an overview of the language, the reader may refer to [13] for more details.

In the impredicative F_ω 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 κ. 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 Ω would then be

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

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

Figure 1: Syntax of the λ_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 λ_i^{P+} types

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 Ω from the kind of the argument of the constructor $\forall_\Omega$. Hence in our new λ_i^{P+} calculus (see Figures 1 and 2), we extend F_ω with variable and polymorphic kinds (χ 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 $\boldsymbol{\forall}\,[\kappa]\,(\lambda\alpha : \kappa.\, \tau)$. Existential quantifiers can be treated in the same way.

While analyzing a polymorphic type, $\boldsymbol{\forall}\,[\kappa]\,\tau$, the kind κ 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

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

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 kind 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 χ in the type τ. The formation rules are shown below (Γ denotes a value environment).

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

Kind formation $\quad \mathcal{E} \vdash \kappa$

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

Type environment formation $\quad \mathcal{E} \vdash \Delta$

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

Type formation $\quad \mathcal{E}; \Delta \vdash \tau \,:\, \kappa$

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

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

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

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

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

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

Term environment formation $\quad \mathcal{E}; \Delta \vdash \Gamma$

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Formation rules of λ_i^{P+}

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

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

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

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

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

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

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

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

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

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

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

Figure 4: Selected λ_i^{P+} type reduction rules

Furthermore, since our goal is fully reflexive type analysis, we need to analyze kind-polymorphic types as well. As with polymorphic types, we can represent the type $\forall^{+}\chi.\,\tau$ as the application of a type constructor $\mathsf{\forall}^{+}$ of kind $(\forall\chi.\,\Omega) \to \Omega$ to the type $\Lambda\chi.\,\tau$.

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 Ω and returns a type of kind Ω (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 τ_{int} branch. At the function type $\tau \to \tau'$, it applies the $\tau_{\to}$ branch to the components τ and τ' and to the result of the iteration over τ and τ'.

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

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 κ' (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 κ'. 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 iteration.

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

Recursive types are formed using the μ 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 corresponding branch of the typecase operator is just a divergent term; this branch will never be used. 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 λ_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 μ-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 μ constructor has a negative occurrence of the kind Ω, this case must be handled specially. Therefore, the Typerec does not act as an iterator for the μ constructor. Instead, it directly analyzes the body of the recursive type. In essence, we have made the μ constructor transparent to the analysis. Operationally, the number of nested μ constructors strictly decreases at every reduction of the Typerec, ensuring termination after a finite number of steps.

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

$$\text{typecase}[\tau]\ \text{int of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow e_{\text{int}}$$

$$\text{typecase}[\tau]\ (\tau_1 \to \tau_2)\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow e_{\to}\ [\tau_1]\ [\tau_2]$$

$$\text{typecase}[\tau]\ (\forall\,[\kappa]\,\tau')\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow e_{\forall}\ [\kappa]^{+}\ [\tau']$$

$$\text{typecase}[\tau]\ (\forall^{+}\tau')\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow e_{\forall^+}\ [\tau']$$

$$\text{typecase}[\tau]\ (\mu\,\tau')\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow e_{\mu}\ [\tau']$$

$$\text{typecase}[\tau]\ (\text{Place}\,\tau')\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu}) \rightsquigarrow \text{typecase}[\tau]\ (\text{Place}\,\tau')\ \text{of}\ (e_{\text{int}};\ e_{\to};\ e_{\forall};\ e_{\forall^+};\ e_{\mu})$$

Figure 5: Selected term reduction rules of λ_i^{P+}

performed using the typecase operator. Since the term level includes a fixed-point operator, typecase is not iterative; it inspects a given type τ' and passes its components to the corresponding branch. The reduction rules for typecase are in Figure 5.

We will give an example of type analysis by coding the polymorphic equality function eq. We will extend the example in [7] to handle quantified types. To illustrate the type level analysis we will use the Typerec operator to define the class of types admitting equality comparisons. We define a type operator $\text{Eq} : \Omega \to \Omega$ which maps function and polymorphic types to the type Void, and require the arguments of eq to be of type $\text{Eq}\,\tau$ for any type τ. (Here $\text{Void} \equiv \forall\alpha : \Omega.\ \alpha$ is a type with no values). To make the example more realistic, we extend the language with a product type constructor ($\times$) of the same kind as ($\to$) and use bool instead of int as the base type. The type analysis constructs operate on the $\times$ constructor in a manner similar to the $\to$ constructor, and operate on the bool type in a manner similar to the int type.

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

$$\begin{array}{l} \text{t} = \lambda\alpha : \Omega.\ \text{Typerec}\ \alpha\ \text{of}\ (\tau_{\text{int}};\ \tau_{\to};\ \tau_{\forall};\ \tau_{\forall^+}) \\ \text{where}\quad \tau_{\to} = \lambda\alpha_1 : \Omega.\ \lambda\alpha_2 : \Omega.\ \lambda\alpha_1' : \kappa.\ \lambda\alpha_2' : \kappa.\ \tau_{\to}' \\ \qquad\qquad \tau_{\forall} = \Lambda\chi.\ \lambda\alpha : \chi \to \Omega.\ \lambda\alpha' : \chi \to \kappa.\ \tau_{\forall}' \\ \qquad\qquad \tau_{\forall^+} = \lambda\alpha : (\forall\chi.\,\Omega).\ \lambda\alpha' : (\forall\chi.\,\kappa).\ \tau_{\forall^+}' \end{array}$$

we write

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

In this syntax the Eq type operator is defined as:

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

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

$$\begin{array}{l} \text{fix eq} : \forall\alpha : \Omega.\ \text{Eq}\,\alpha \to \text{Eq}\,\alpha \to \text{bool}. \\ \quad = \Lambda\alpha : \Omega. \\ \qquad \text{typecase}[\lambda\gamma : \Omega.\ \text{Eq}\,\gamma \to \text{Eq}\,\gamma \to \text{bool}]\ \alpha\ \text{of} \\ \qquad\quad \text{bool} \Rightarrow \lambda\text{x} : \text{bool}.\ \lambda\text{y} : \text{bool}.\ \text{primEqBool}\,\text{x}\,\text{y} \\ \qquad\quad \times \quad \Rightarrow \Lambda\beta_1 : \Omega.\ \Lambda\beta_2 : \Omega. \\ \qquad\qquad\qquad \lambda\text{x} : \text{Eq}\,(\beta_1) \times \text{Eq}\,(\beta_2).\ \lambda\text{y} : \text{Eq}\,(\beta_1) \times \text{Eq}\,(\beta_2). \\ \qquad\qquad\qquad\qquad \text{eq}\,[\beta_1]\,(\text{x}.1)\,(\text{y}.1)\ \text{and}\ \text{eq}\,[\beta_2]\,(\text{x}.2)\,(\text{y}.2) \\ \qquad\quad \to \quad \Rightarrow \ldots \\ \qquad\quad \forall \quad \Rightarrow \ldots \\ \qquad\quad \forall^{+} \quad \Rightarrow \ldots \\ \qquad\quad \mu \quad \Rightarrow \Lambda\beta : \Omega \to \Omega.\ \lambda\text{x} : \mu\,(\lambda\alpha' : \Omega.\ \text{Eq}\,(\beta\,(\text{Place}\,\alpha'))). \\ \qquad\qquad\qquad \lambda\text{y} : \mu\,(\lambda\alpha' : \Omega.\ \text{Eq}\,(\beta\,(\text{Place}\,\alpha'))). \\ \qquad\qquad\qquad \text{eq}\,[\alpha_1]\,(\text{unfold}[\alpha_2]\,\text{x})\,(\text{unfold}[\alpha_2]\,\text{y}) \\ \qquad\quad \text{where} \\ \qquad\qquad \alpha_1 = \beta\,(\text{Place}\,(\mu\,(\lambda\alpha' : \Omega.\ \text{Eq}\,(\beta\,(\text{Place}\,\alpha'))))) \\ \qquad\qquad \alpha_2 = \lambda\alpha' : \Omega.\ \text{Eq}\,(\beta\,(\text{Place}\,\alpha')) \end{array}$$

Figure 6: Polymorphic equality in λ_i^{P+}

The polymorphic equality function eq is shown in Figure 6. The function and polymorphic cases are omitted as they will never be executed (since the domain type is restricted to types of the form $\text{Eq}\,\tau$).

The language λ_i^{P+} has the following properties. The proofs are similar to the proofs for the language λ_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.*

We say that a term is *stuck* if it is not a value and it is not related to another term by the relation of reduction ($\rightsquigarrow$).

Proposition 3.3 (Type Safety) *If* $\vdash e : \tau$ *and* $e \rightsquigarrow^{*} e'$ *then* e' *is not stuck.*

3.1 Type analysis in λ_i^{P+}

In our previous work [13], we proposed the language λ_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 λ_i^{P+} is restricted in two ways. First, the Typerec operator must return a type of kind Ω. 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 Ω, the result of the Typerec must also be of kind Ω. Thus, in practice, a Typerec will be employed to form types of kind Ω.

In some cases a Typerec is used to enforce typing constraints—for example, in the case of polymorphic equality

$$
\begin{array}{lrcl}
(\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 λ_R^P language

above, a Typerec was used to express the constraint that the set of equality types does not include function 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.

4 The target language λ_R^P

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

4.1 The analyzable components in λ_R^P

In λ_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 Ω includes the set of types, while the kind T includes the set of tags. For every constructor that generates a type of kind Ω, we have a corresponding constructor that generates a tag of kind T; for example, int corresponds to T_{int} and $\twoheadrightarrow$ corresponds to $T_{\rightarrow}$. The type analysis construct at the type level is Tagrec and operates only on tags.

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_{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 λ_R^P includes a unary type constructor R of kind $\mathsf{T} \rightarrow \Omega$ to type the term level representations. Given a tag τ (of kind T), the term representation of τ 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 τ [3].

If the tag τ is of a function kind $\kappa \rightarrow \kappa'$, then the representation of τ 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 τ is of a variable kind χ. The only way of knowing the type of its representation R_{χ} is to construct it when χ is instantiated. Hence programs translated into λ_R^P must be such that for every kind variable χ in the program, a corresponding type variable α_{χ}, representing the type of the term representation for a tag of kind χ, is also available.

This is why we need to extend the CWM framework. Their source language does not include kind polymorphism; therefore, 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 constructors and split the type calculus into types and tags. Consider the $\forall$ and the $\forall^{+}$ type constructors in λ_i^{P+}. The $\forall$ constructor binds a kind. When it is translated into λ_R^P, the corresponding constructor must also, in addition, bind a type. Therefore, we need the new constructor $T_{\forall}$. Similarly, the $\forall^{+}$ constructor binds a kind function. When it is translated into λ_R^P, the function bound by the corresponding constructor must not only bind a kind, but a type as well. Therefore, we introduce the new constructor $T_{\forall^+}$. Furthermore, if we only have Ω as the primitive kind, it will no longer be inductive. (The inductive definition would break for $T_{\forall^+}$). Therefore, we introduce a new kind T for tags, and allow analysis only over tags.

This leads us to the kind translation from λ_i^{P+} to λ_R^P (Figure 8). Since the analyzable component of λ_R^P is of kind T, the λ_i^{P+} kind Ω is mapped to T. The translation of a polymorphic kind $\forall \chi.\, \kappa$ takes into account that the representation type for the kind variable χ will be provided by a new type argument of kind $\chi \rightarrow \Omega$. The following lemma holds for the kind translation:

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

Proof By induction over the structure of κ. □

Figure 9 shows the function R_{κ}, the type of the term representation for a λ_i^{P+} type of kind κ. Since this function is used by the translation from λ_i^{P+} to λ_R^P, it is defined by induction on λ_i^{P+} kinds.

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

Figure 8: Translation of λ_i^{P+} kinds to λ_R^{P} kinds

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

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

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

Figure 9: Types of representations at higher kinds

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

Proof By induction over the structure of κ. □

The formation rules for tags are displayed in Figure 10. Since the translation maps λ_i^{P+} type constructors to these tags, a type constructor of kind κ 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 types of the term representations of primitive type constructors. These types agree with the definition of the function R_κ; for example, the type of $\mathsf{R}_\to$ is $R_{\Omega\to\Omega\to\Omega}\,(T_\to)$.

4.3 Tag analysis in λ_R^P

We now consider the tag analysis constructs in more detail. The term level analysis is done by the repcase construct. Figure 10 and Figure 12 show its static and dynamic semantics respectively. The expression being analyzed must be of type $R\,\tau$; therefore, 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

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

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

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

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

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

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

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

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

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

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

Figure 10: Formation rules for the new constructs in λ_R^P

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

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

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

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

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

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

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

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

Figure 11: Reduction rules for λ_R^P Typerec

Figure 11 shows the reduction rules for the Tagrec, which are similar to the reduction rules for the source language 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

$$\begin{array}{l}\mathsf{Tagrec}\ (T_{\rightarrow}\ \tau\ \tau')\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow\\ \quad \tau_{\rightarrow}\ \tau\ \tau'\ (\mathsf{Tagrec}\ \tau\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}))\\ \qquad\quad (\mathsf{Tagrec}\ \tau'\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}))\end{array}$$

Similarly, the reduction for the polymorphic tag is

$$\begin{array}{l}\mathsf{Tagrec}\ (T_{\forall}\ [\kappa]\ \tau_\kappa\ \tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}) \rightsquigarrow\\ \tau_{\forall}\ [\kappa]\ \tau_\kappa\ \tau\ (\lambda\alpha\!:\!\kappa.\,\mathsf{Tagrec}\ (\tau\ \alpha)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+};\ \tau_{R}))\end{array}$$

Recursive tags are handled in a manner similar to recursive types in λ_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 Tagrec.

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

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

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

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

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

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

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

Figure 12: Selected term reduction rules of λ_R^P

The reduction rules also include a rule for the Pl constructor. The Pl constructor is used to handle recursive tags in the F function (Section 4.4). This constructor is again an implementation artifact in λ_R^P and has no counterpart in the source language. Its reduction rule will never be used in a program translated from λ_i^{P+}.

4.4 The tag interpretation function

Programs in λ_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 Ω, this function must map tags of kind T to types of kind Ω. This implies that we can use an iterator over tags to define the function as follows:

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

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

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

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

Figure 13: Translation of λ_i^{P+} types to λ_R^P tags

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 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 F has free kind variables. □

The language λ_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 $\vdash e:\tau$ and $e \leadsto^* e'$, then e' is not stuck.*

5 Translation from λ_i^{P+} to λ_R^P

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

Figure 13 shows the translation of λ_i^{P+} types into λ_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 must maintain two invariants. First, for every accessible kind variable χ, it adds a corresponding type variable α_χ; this variable gives the type of the term representation for a tag of kind χ. At every kind application, the translation uses the function R_κ (Figure 9) to compute this type. Thus, the translations of kind abstractions and kind applications are

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

$$
\begin{array}{rl}
|i| &= i \\
|x| &= x \\
|\Lambda^{+}\chi.\,v| &= \Lambda^{+}\chi.\,\Lambda\alpha_\chi:\chi\to\Omega.\,|v| \\
|e\,[\kappa]^{+}| &= |e|\,[|\kappa|]^{+}\,[R_\kappa] \\
|\Lambda\alpha:\kappa.\,v| &= \Lambda\alpha:|\kappa|.\,\lambda x_\alpha:R_\kappa\,\alpha.\,|v| \\
|e\,[\tau]| &= |e|\,[|\tau|]\,\Re(\tau) \\
|\lambda x:\tau.\,e| &= \lambda x:\mathsf{F}\,|\tau|.\,|e| \\
|e\,e'| &= |e|\,|e'| \\
|\mathsf{fix}\,x:\tau.\,v| &= \mathsf{fix}\,x:\mathsf{F}\,|\tau|.\,|v| \\
|\mathsf{fold}[\tau]\ e| &= \mathsf{fold}[|\tau|]\ |e| \\
|\mathsf{unfold}[\tau]\ e| &= \mathsf{unfold}[|\tau|]\ |e|
\end{array}
$$

$$
\begin{array}{l}
|\mathsf{typecase}[\tau]\ \tau'\ \mathsf{of}\ (e_{\mathsf{int}};\ e_{\rightarrow};\ e_{\forall};\ e_{\forall^{+}};\ e_{\mu})| \\
\quad = \mathsf{repcase}[\lambda\alpha:\mathsf{T}.\,\mathsf{F}\,(|\tau|\,\alpha)]\ \Re(\tau')\ \mathsf{of} \\
\qquad \mathsf{R}_{\mathsf{int}} \Rightarrow |e_{\mathsf{int}}| \\
\qquad \mathsf{R}_{\rightarrow} \Rightarrow |e_{\rightarrow}| \\
\qquad \mathsf{R}_{\forall} \Rightarrow |e_{\forall}| \\
\qquad \mathsf{R}_{\forall^{+}} \Rightarrow |e_{\forall^{+}}| \\
\qquad \mathsf{R}_{R} \Rightarrow \Lambda\beta:\mathsf{T}.\,\lambda x:R\,\beta.\,\mathsf{fix}\,x:\mathsf{F}\,(|\tau|\,(T_R\,\beta)).\,x \\
\qquad \mathsf{R}_{\mu} \Rightarrow |e_{\mu}| \\
\qquad \mathsf{R}_{pl} \Rightarrow \Lambda\beta:\mathsf{T}.\,\lambda x:R\,\beta.\,\mathsf{fix}\,x:\mathsf{F}\,(|\tau|\,(T_{pl}\,\beta)).\,x
\end{array}
$$

Figure 14: Translation of λ_i^{P+} terms to λ_R^P terms

Second, for every accessible type variable α, a term variable x_α is introduced, providing the corresponding term representation of α. 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 a representation. 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 type interpretation function 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 Typerec. Since Typerec is recursive, we use a combination of a repcase and a fix. We will illustrate only one case here; the other cases can be reasoned about similarly.

Consider the reduction of $\mathsf{Ty}\,(\tau' \to \tau'')$, 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 τ', τ'', and the result of the recursive calls to the Typerec. The representations of τ' and τ'' are bound to the variables x_α and x_β; by assumption the representations for the results of the recursive calls are obtained from the recursive calls to the function f.

In the following propositions the original λ_i^{P+} type environment Δ is extended with a type environment $\Delta(\mathcal{E})$ which binds

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

$$
\begin{array}{l}
\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_{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\quad \Re(\tau_\rightarrow)\,[\alpha]\,(x_\alpha)\,[\beta]\,(x_\beta) \\
\qquad\qquad\qquad [\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\quad \Re(\tau_\forall)\,[\chi]^{+}\,[\alpha_\chi]\,[\alpha]\,(x_\alpha)\,[\lambda\beta:\chi.\,\tau^*\,(\alpha\,\beta)] \\
\qquad\qquad\qquad (\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\quad \Re(\tau_{\forall^{+}})\,[\alpha]\,(x_\alpha)\,[\Lambda\chi.\,\lambda\alpha_\chi : \chi \rightarrow \Omega.\, \tau^*\,(\alpha\,[\chi]\,\alpha_\chi)] \\
\qquad\qquad\quad (\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\quad \mathsf{R}_\mu\,[\lambda\beta:\mathsf{T}.\,\tau^*\,(\alpha\,(T_{pl}\,\beta))] \\
\qquad\qquad\quad (\Lambda\beta:\mathsf{T}.\, \lambda x_\beta : R\,\beta. \\
\qquad\qquad\qquad \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{array}
$$

Figure 15: Representation of λ_i^{P+} types as λ_R^P terms

a type variable α_χ of kind $\chi \rightarrow \Omega$ for each $\chi \in \mathcal{E}$. Similarly the term-level translations extend the term environment Γ with $\Gamma(\Delta)$, binding a variable x_α of type $R_\kappa\,\alpha$ for each type variable α bound in Δ with kind κ.

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

Proof Follows directly by induction over the structure of τ. □

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

Proof By induction over the structure of τ. 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* λ_i^{P+}*, then* $|\mathcal{E}|; |\Delta|,\ \Delta(\mathcal{E}); |\Gamma|,\ \Gamma(\Delta) \vdash |e| : \mathsf{F}\,|\tau|$ *holds in* λ_R^P.

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

6 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 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 λ-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 untyped language

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

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

Proposition A.1 *If $e \leadsto^* e_1$, then $e^\circ \leadsto^* {e_1}^\circ$.*

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

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}_{\to}}^\circ &= \mathsf{R}_{\to} \\
(\mathsf{R}_{\to}\, [\tau])^\circ &= \mathsf{R}_{\to}\, 1 \\
(\mathsf{R}_{\to}\, [\tau]\, e)^\circ &= \mathsf{R}_{\to}\, 1\, e^\circ \\
(\mathsf{R}_{\to}\, [\tau]\, e\, [\tau'])^\circ &= \mathsf{R}_{\to}\, 1\, e^\circ\, 1 \\
(\mathsf{R}_{\to}\, [\tau]\, e\, [\tau']\, e_1)^\circ &= \mathsf{R}_{\to}\, 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}
$$

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

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