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

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 all types of a higher order typed source language, including quantified types, into a system that can propagate type information through all compilation phases.

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 the types of arbitrary source language terms, which we refer to as *fully reflexive type analysis*.

On the other hand, generation of certified code [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 types of arbitrary source language terms) 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 existing term operations to process 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 only the analysis of types with no binding structure; 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, which requires constraining the analysis of recursive types in the source language, and introducing unconventional fold and unfold constructs 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 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.

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

[*]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.

Typerec for the type level. For example, a polymorphic subscript function for arrays might be written as the following pseudo-code:

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

Here 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 intarray and 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 typecase analysis at the term level. The subscript operation would then be typed as

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

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

3 The source language λ_i^{P+}

To illustrate our ideas, we define the λ_i^{P+} calculus with syntax shown in Figures 1 and 2. The static semantics of λ_i^{P+} uses the following three environments:

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

It can be shown that the formation rules in Figure 3 enforce the requirement that the environments are well-formed, and moreover, all inferred 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 $\mathcal{E};\Delta \vdash \tau\ :\ \Omega$. Reduction in the type language is defined according to the rules in Figure 4. The reduction rules for the term level type analysis construct typecase can be found in Figure 5.

The language λ_i^{P+} extends the language λ_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 calculus F_ω 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

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

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

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

Figure 2: Syntactic sugar for λ_i^{P+} types

constructors that can generate types of kind Ω would then be

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

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 occurrences of Ω from the kind of the argument of some constructors, *e.g.* $\forall_\Omega$; these occurrences make Ω non-inductive, so that defining a Typerec-like "iterator" over Ω would break the crucial strong normalization property of the type language. Hence in our λ_i^{P+} calculus 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 $\boldsymbol{\forall}\,[\kappa]\,(\lambda\alpha\!:\!\kappa.\,\tau)$.

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 Typerec operator needs a kind abstraction in the branch corresponding to the $\boldsymbol{\forall}$ constructor. We provide kind abstraction $\Lambda\chi.\,\tau$ 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 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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 3: Formation rules of λ_i^{P+}

$\boxed{\text{Type reduction} \quad \mathcal{E};\Delta \vdash \tau_1 \leadsto \tau_2 : \kappa}$

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

$$\frac{\mathcal{E};\Delta \vdash \tau : \kappa \to \kappa' \quad \alpha \notin \mathit{ftv}(\tau)}{\mathcal{E};\Delta \vdash \lambda\alpha{:}\kappa.\,\tau\,\alpha \leadsto \tau : \kappa \to \kappa'} \qquad \frac{\mathcal{E};\Delta \vdash \tau : \forall\chi'.\,\kappa \quad \chi \notin \mathit{fkv}(\tau)}{\mathcal{E};\Delta \vdash \Lambda\chi.\,\tau\,[\chi] \leadsto \tau : \forall\chi'.\,\kappa}$$

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

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

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

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

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

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

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

typecase must bind a kind variable to the kind of the quantified type variable. For that purpose we introduce kind abstraction $\overset{+}{\Lambda}\chi.\,v$ 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 $\overset{+}{\forall}\chi.\,\tau$ that binds the kind variable χ in the type τ. The formation rules are shown below.

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

Furthermore, since our goal is fully reflexive type analysis, we need to analyze kind-polymorphic types as well. As with polymorphic types, we can represent the type $\overset{+}{\forall}\chi.\,\tau$ as the application of a type constructor $\boldsymbol{\forall}^{+}$ 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 results of recursively processing τ and τ'.

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

When analyzing a polymorphic type, the reduction rule is

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

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 variable kind, instantiated to κ' during the analysis. 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

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

The $\boldsymbol{\forall}^{+}$-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.

The treatment of recursive types is similar to that in the language λ_i^{Q} of [13], but simplified. They are formed using the $\mathsf{\mu}$ constructor of kind $(\Omega \to \Omega) \to \Omega$. Following ideas due to Fegaras and Sheard [6], 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; the term language provides no constructors to create a non-variable object of type $\mathsf{Place}\,\tau$ for any τ.

The simpler kind language of λ_i^{P+} (in comparison with λ_i^{Q}) comes at the price of restricting the result of the analysis of recursive types by a Typerec to always be a recursive type. Thus we avoid a problem arising when the analysis of a recursive type yields a result unrelated to the analysis of its unfolding, described further in Section 4.4.

Since the argument of the $\mathsf{\mu}$ constructor has a negative occurrence of the kind Ω, this case must be handled differently.

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

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

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

Figure 6: The function toString

Typerec does not act as an iterator for the μ constructor. Instead, it 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 such a type variable.

$$\begin{aligned}
&\mathsf{Typerec}\ (\mu\,\tau)\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}) \leadsto\\
&\mu\,(\lambda\alpha:\Omega.\ \mathsf{Typerec}\ (\tau\ (\mathsf{Place}\,\alpha))\ \mathsf{of}\ (\tau_{\mathsf{int}};\ \tau_{\rightarrow};\ \tau_{\forall};\ \tau_{\forall^+}))
\end{aligned}$$

In essence, we have made the μ constructor transparent to the analysis. Operationally, the number of nested μ constructors in the type analyzed by a Typerec strictly decreases at every reduction involving μ, ensuring termination after a finite number of steps.

The term expressions are mostly standard. We use the standard fold and unfold constructs 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 τ' and passes its constituents to the corresponding branch. The reduction rules for typecase are in Figure 5.

Existential types can be handled similarly 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 $\exists\,[\kappa]\,(\lambda\alpha:\kappa.\,\tau)$. Typerec and typecase are augmented with $\tau_{\exists}$ and $e_{\exists}$ branches respectively. The reduction rules are exactly analogous to those for the polymorphic type.

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. The type operator $\mathsf{Eq} : \Omega \rightarrow \Omega$, defined below, 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 ($\times\!\!\!\times$) of the same kind as ($\twoheadrightarrow$). The type analysis constructs operate on the $\times\!\!\!\times$ 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

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

we write

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

In this syntax the Eq type operator is defined as:

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

where the last line of the definition is not under programmer control.

As an example of the term level analysis in λ_i^{P+}, consider the function toString shown in Figure 6. This function uses the type of a value to produce its string representation; we assume having a nullary type constructor string in the language. The primitive function intToString converts an integer to its string representation, and use ˆ to denote string concatenation.

The language λ_i^{P+} has the following properties, with proofs similar to those 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.*

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

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

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

Other applications of type analysis also follow this pattern. Consider a copying garbage collector [14]. Its copying function would use a $\mathsf{Typerec}$ 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.

Our language allows the analysis of recursive types within both the term language and the type language, but combining them is subject to severe limitations. For instance, one can write a polymorphic printer that analyses types at runtime, and 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 $\forall\alpha:\Omega.\, \mathsf{Eq}\,\alpha \to \mathsf{Eq}\,\alpha \to \mathsf{bool}$. 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, no useful term can be provided in the corresponding branch of $\mathsf{typecase}$. This problem does not affect the μ-free segment of the language and its translation.

4 The target language λ_R^P

Figure 7 shows the syntax of the λ_R^P language, the target of our translation, which reflects type information at the term level in preparation for type erasure. To make the presentation simpler, we will describe many of the features of λ_R^P in the context of the translation from λ_i^{P+}.

$$
\begin{array}{lrl}
(kinds) & \kappa ::= & \Omega \mid \mathsf{T} \mid \kappa \to \kappa' \mid \chi \mid \forall\chi.\,\kappa \\[1em]
(types) & \tau ::= & \mathsf{int} \mid \twoheadrightarrow \mid \forall \mid \overset{+}{\forall} \mid \mu \mid \mathsf{Pl} \mid R \\
 & & \mid T_{\mathsf{int}} \mid T_{\to} \mid T_{\forall} \mid T_{\overset{+}{\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_{\to};\ \tau_{\forall};\ \tau_{\overset{+}{\forall}};\ \tau_R) \\[1em]
(values) & v ::= & i \mid \overset{+}{\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}_{\to} \mid \mathsf{R}_{\to}\,[\tau] \mid \mathsf{R}_{\to}\,[\tau]\,v \\
 & & \mid \mathsf{R}_{\to}\,[\tau]\,v\,[\tau'] \mid \mathsf{R}_{\to}\,[\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}_{\overset{+}{\forall}} \mid \mathsf{R}_{\overset{+}{\forall}}\,[\tau] \mid \mathsf{R}_{\overset{+}{\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 \\[1em]
(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_{\to};\ e_{\forall};\ e_{\overset{+}{\forall}};\ e_R;\ e_{\mu};\ e_{pl})
\end{array}
$$

Figure 7: Syntax of the λ_R^P language

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 language to distinguish between the two: Kind Ω is assigned to types, while kind T is assigned to tags. For every constructor yielding a type of kind Ω we have a corresponding constructor that generates a tag of kind T; for example, T_{int} corresponds to int and $T_{\to}$ corresponds to $\twoheadrightarrow$. The type analysis construct at the type level is Tagrec, which operates only on tags.

At the term level, we add representations for tags. The term level operator (now called $\mathsf{repcase}$) 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 $\mathsf{T} \to \Omega$ 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].

The functionality of R is generalized at higher kinds by R_κ, a type function of kind $\kappa \to \Omega$, such that $R_\kappa\,\tau$ is the type of the term representation for type τ of kind κ. For instance, if the tag τ is of a function kind $\kappa \to \kappa'$, then the term representation of τ is

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

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

a polymorphic function from representations to representations:

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

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.

In comparison, the source language of CWM [3] does not include kind polymorphism, which means that the types of all representations are known statically. We need to extend the framework with types of representations of variable kinds.

Consider for instance the type $\forall\,[\kappa]\,\tau$ in λ_i^{P+}. The $\forall$ branch of a typecase construct must reduce to an abstraction $\overset{+}{\Lambda}\chi.\,\Lambda\alpha\!:\!\chi \to \Omega.\,e$. After translation to λ_R^P, in order to compute the type of the representation of α in e, we need to know the type of the representations of types of kind χ. Therefore this type must be passed as an extra argument to the $\forall$ branch, which means it must be "packed" together with κ and τ using the translated $\forall$ constructor. Thus, if mapped to a constructor for kind Ω in λ_R^P, its kind would be $\forall\chi.\,(\chi \to \Omega) \to (\chi \to \Omega) \to \Omega$, adding a parameter of kind $\chi \to \Omega$ associated with the kind variable χ. A similar situation arises with $\overset{+}{\forall}$, only this time with an unpleasant twist: The kind of the translated constructor must be $(\forall\chi.\,(\chi \to \Omega) \to \Omega) \to \Omega$, in which there is a double-negative occurrence of Ω, making Ω non-inductive.

To preserve the inductive structure of the kinds, we split the type calculus into types and tags. The new constructor $T_{\overset{+}{\forall}}$ is of kind $(\forall\chi.\,(\chi \to \Omega) \to \mathsf{T}) \to \mathsf{T}$, which does not suffer from negative occurrences since Ω is defined independently of T. Type analysis is restricted to tags since they carry the information needed to reconstruct the types of representations.

This leads us to the kind translation from λ_i^{P+} to λ_R^P (Figure 8). Since the analysis in λ_R^P is on kind T, the λ_i^{P+} kind Ω is mapped to T. The polymorphic kind $\forall\chi.\,\kappa$ is translated to $\forall\chi.\,(\chi \to \Omega) \to |\kappa|$. Note that every kind variable χ must now have a corresponding type variable α_χ of kind $\chi \to \Omega$, providing the type of term representations for types of kind χ.

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

Proof By induction over the structure of κ. □

Figure 9 shows the function R_κ. Suppose τ is a λ_i^{P+} type of kind κ and $|\tau|$ is its translation into λ_R^P. The function R_κ gives the type of the term representation of $|\tau|$. 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.

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

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

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

Figure 9: Types of representations at higher kinds

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

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. Figures 10 and 11 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 selects a branch according to the top constructor of the representation, and passes the components of the representation to it.

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+}).

Figure 12 shows the reduction rules for the Tagrec, which are similar to the reduction rules for the source language Typerec: given a tag, it recurses on the components of the tag and then passes the result of the recursive calls, along with the original components, to the corresponding branch. 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.

The reduction rules also include a rule for the Pl constructor. The Pl constructor is used to handle recursive tags in the F func-

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Figure 11: Selected term reduction rules of λ_R^P

tion (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 be defined using a function mapping tags to types. Since these annotations are always of kind Ω, this function must be of kind $\mathsf{T} \to \Omega$. We can use an iterator over tags to define the function as follows:

$$\begin{array}{lll} \mathsf{F}\,(T_{\mathsf{int}}) & = & \mathsf{int} \\ \mathsf{F}\,(T_{\to}\,\tau_1\,\tau_2) & = & \mathsf{F}\,(\tau_1) \to \mathsf{F}\,(\tau_2) \\ \mathsf{F}\,(T_{\forall}\,[\chi]\,\alpha_\chi\,\tau) & = & \forall\alpha\!:\!\chi.\,\alpha_\chi\,\alpha \to \mathsf{F}\,(\tau\,\alpha) \\ \mathsf{F}\,(T_{\forall^+}\,\tau) & = & \forall\chi.\,\forall\alpha_\chi\!:\!\chi \to \Omega.\,\mathsf{F}\,(\tau\,[\chi]\,\alpha_\chi) \\ \mathsf{F}\,(T_{\mu}\,\tau) & = & \mu(\lambda\alpha\!:\!\Omega.\,\mathsf{F}\,(\tau\,(\mathsf{Pl}\,\alpha))) \\ \mathsf{F}\,(\mathsf{Pl}\,\tau) & = & \tau \\ \mathsf{F}\,(T_{R}\,\tau) & = & \mathsf{int} \\ \mathsf{F}\,(T_{pl}\,\tau) & = & \mathsf{int} \end{array}$$

The function F takes a type of kind T and converts it to the corresponding type of kind Ω. The branches for the T_R and the T_{pl} tags 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.

The type interpretation function has the following properties.

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

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

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

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

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

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

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

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

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

Figure 12: Reduction rules for λ_R^P Typerec

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* $\vdash e:\tau$, *then either* e *is a value, or there exists a term* e' *such that* $e \leadsto e'$ *and* $\vdash e':\tau$.

Note that the rules for fold and unfold in Figure 10 unfold a recursive type (of kind T) under the tag interpretation function. If we allowed a Typerec, and therefore a Tagrec, to have user-defined result for the analysis of recursive types, this would have allowed one to write type functions like

$$\tau = \lambda\alpha:\mathsf{T}.\ \mathsf{Tagrec}\ \alpha\ \mathsf{of}\ (\ldots;\ T_\mu \Rightarrow T_{\mathsf{int}})$$

with the property that $\mathsf{F}\ (\tau\ (T_\mu\ \tau)) = \mathsf{int}$, but $\mathsf{F}\ (T_\mu\ \tau) = \mu\ (\lambda\alpha:\Omega.\ \alpha)$, breaking the type safety theorem.

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

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

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

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 tags.

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 kind variable χ in scope, 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

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

Second, for every type variable α in scope, 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 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 F to map tags to types.

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

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

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

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 $\mathsf{Typerec}\;\tau\;\mathsf{of}\;(\tau_{\mathsf{int}};\; \tau_{\to};\; \tau_{\forall};\; \tau_{\forall^{+}})$. This type reduces to $\tau_{\to}\,\tau'\,\tau''\,(\mathsf{Ty}\,(\tau'))\,(\mathsf{Ty}\,(\tau''))$. Therefore, in the translation, the term representation of $\tau_{\to}$ 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+} kind environment Δ is extended with a kind environment $\Delta(\mathcal{E})$ which binds a type variable α_{χ} of kind $\chi \to \Omega$ for each $\chi \in \mathcal{E}$. Similarly the term-level translations extend the type 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}*.*

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

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

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

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. □

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

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

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

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

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

6 The untyped language

This section shows that in λ_R^P 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 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 λ_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]. They consider a language with analyzes over types with no binding structure. 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 powerful 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 corresponding to the dictionary for that type (for some class). 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; while the handling of polymorphic and existential types appears adequate, problems remain open in the treatment of recursive types in our source language. The framework does not rely on 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.

Acknowledgements

We are grateful to the anonymous referees for their insightful comments and suggestions on improving the presentation.

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.