

Intensional Polymorphism in Type-Erasure Semantics

KARL CRARY,

Carnegie Mellon University

STEPHANIE WEIRICH and GREG MORRISSETT

Cornell University

Abstract

Intensional polymorphism, the ability to dispatch to different routines based on types at run time, enables a variety of advanced implementation techniques for polymorphic languages, including tag-free garbage collection, unboxed function arguments, polymorphic marshalling, and flattened data structures. To date, languages that support intensional polymorphism have required a type-passing (as opposed to type-erasure) interpretation where types are constructed and passed to polymorphic functions at run time. Unfortunately, type-passing suffers from a number of drawbacks: it requires duplication of run-time constructs at the term and type levels, it prevents abstraction, and it severely complicates polymorphic closure conversion.

We present a type-theoretic framework that supports intensional polymorphism, but avoids many of the disadvantages of type passing. In our approach, run-time type information is represented by ordinary terms. This avoids the duplication problem, allows us to recover abstraction, and avoids complications with closure conversion. In addition, our type system provides another improvement in expressiveness; it allows unknown types to be refined in place, thereby avoiding certain beta-expansions required by other frameworks.

1 Introduction

Type-directed compilers use type information to enable optimizations and transformations that are difficult or impossible without such information (Leroy, 1992; Harper & Morrisett, 1995; Morrisett, 1995; Birkedal *et al.*, 1996; Ruf, 1997; Shao, 1997a). However, type-directed compilers for some languages such as and ML face the difficulty that some type information cannot be known at compile time. For example, polymorphic code in ML may operate on inputs of type α where α is not only unknown, but may in fact be instantiated by a variety of different types.

In order to use type information in contexts where it cannot be provided statically, a number of advanced implementation techniques process type information at run time (Harper & Morrisett, 1995; Morrisett, 1995; Tolmach, 1994; Morrisett & Harper, 1997; Shao, 1997a). Such type information is used in two ways: behind the scenes, typically by tag-free garbage collectors (Tolmach, 1994; Aditya & Caro, 1993), and explicitly in program code, for a variety of purposes such as efficient

data representation and marshaling (Morrisett, 1995; Harper & Morrisett, 1995; Shao, 1997b). In this paper we focus on the latter area of applications.

To lay a solid foundation for programs that analyze types at run time, Harper and Morrisett devised an internal language, called λ_i^{ML} , which supports the first-class *intensional analysis*¹ of type information (following earlier work by Constable (Constable, 1982; Constable & Zlatin, 1984)). The λ_i^{ML} language and its derivatives were then used extensively in the high-performance ML compilers TIL/ML (Tarditi *et al.*, 1996; Morrisett *et al.*, 1996) and FLINT (Shao, 1997b). Type constructors may be analyzed by “typecase” operators in both the term and the type constructor languages; these operators allow computations and type expressions to depend upon the values of other type expressions at run time.

Supporting intensional type analysis (and the use of type information at run time in general) seems to require semantics where type information is formed and passed to polymorphic functions during computation. However, there are three significant reasons why such a type-passing semantics is unattractive:

- A type-passing language such as λ_i^{ML} requires that type information *always* be constructed and passed to polymorphic functions, even when one does not desire to do so. For example, passing type information at run time comes with a cost, and the type-passing framework cannot express the elimination of that information where appropriate to optimize performance. Also, one may wish to withhold run-time type information from a function to enforce type abstraction, but this is impossible in the type-passing framework.
- Because both terms and type constructors describe run-time execution, type passing results in considerable complexity in language semantics, as a number of run-time semantic devices must be duplicated for both terms and type constructors. Although this duplication does not induce substantial complexity in the substitution-based semantics of λ_i^{ML} , it does as one attempts to give λ_i^{ML} a semantics more faithful to real machines. For example, in semantics that make memory allocation explicit (Morrisett *et al.*, 1995) a central device is a formal heap in which data is stored; in a type-erasure framework one such heap suffices, but when types are passed it is necessary to add a second heap (Morrisett & Harper, 1997), and all the attendant machinery, for type data.

Type passing also greatly complicates low-level intermediate languages, due to the need to support mixed-phase devices (constructs with both type constructor and term level components). This can pose a serious problem for typed intermediate languages, because these devices can disrupt the essential symmetries on which elegant type systems depend. For example, a type-passing semantics for Typed Assembly Language (Morrisett *et al.*, 1999) would require additional instructions for allocating and initializing type constructors, which in turn requires the typing machinery for allocation and initialization to be lifted an additional level into the kind structure.

¹ Type analysis is “intensional” when types are analyzed by their structure, rather than by what terms they contain.

- As a particularly important example of the second issue, type passing severely complicates typed closure conversion (compare the type-passing system of Minamide *et al.* (1996) to the type-erasure system of Morrisett *et al.* (1999)). In a type-erasure framework, the partial application of a polymorphic function to a type may still be considered a value (since the application has no run-time significance), which means that closed code may simply be instantiated with its type environment when a closure is created. In a type-passing framework, the instantiation with a type environment can have some run-time effect, so it must be delayed until the function is invoked. Consequently, closures must include a type environment, necessitating complicated mechanisms including abstract kinds and translucent types (Minamide *et al.*, 1996).

A possible solution to the first problem (but not the second or third) would be to introduce a phase distinction between type constructors: Those purely necessary for type checking would be marked static and the remainder dynamic, with restrictions prohibiting dynamic type information from depending on static type constructors. A framework of how to construct such a language appears in Abadi, *et al.* (1999). A possible solution to the second problem (but not the first or third) would be to combine the type and term languages together in the same syntactic class, as in Pure Type Systems (Barendregt, 1992). However, then the constructs used to describe run-time execution would also complicate compile-time type checking.

In this paper we propose a typed calculus, called λ_R , that ameliorates all three problems of type passing without sacrificing intensional type analysis. The fundamental idea behind our approach is to move the dynamic aspect of the type information from the level of types to the level of ordinary terms. This works by constructing and passing *values* that represent types instead of the types themselves. The connection between a type constructor τ and its term representation v is made in the static semantics by assigning v the special type $R(\tau)$. Semantically, we may interpret $R(\tau)$ as a singleton type that contains only the representation of τ .

This framework resolves the difficulties with type-passing semantics discussed above. In particular, as representations of types are simply terms, we can use the pre-existing term operations to deal with run-time type information in languages and their semantics. Furthermore, we can eliminate the difficulties associated with polymorphic closure conversion, as we show in Section 5. Finally, by making dynamic type information explicit and separable from types, our approach enables the choice *not* to pass representations. In turn, this choice allows us to eliminate the overhead of constructing and passing representations of types where it is not necessary. Current type-passing compilers, such as TIL/ML, already perform this optimization by using annotations that mark whether a type must be passed at run-time. However, these types may not be eliminated until late, untyped phases of the compiler. Our system provides a formal, typed basis for that mechanism.

Perhaps more importantly, the ability not to pass types allows abstraction and parametricity to be recovered. In most type systems, abstraction may be achieved by hiding the identity of types either through parametric polymorphism (Reynolds,

1983) or through existential types (Mitchell & Plotkin, 1988). However, when all types are passed and may be analyzed (as in λ_i^{ML}), the identity of types cannot be hidden and consequently abstraction is impossible. In contrast, a λ_R type can be analyzed only when its representation is available at run time, so abstraction can be achieved simply by not supplying type representations.

For example, consider the type $\exists\alpha.\alpha$. When all types may be analyzed, this type implements a *dynamic* type; an expression of this type provides an object of some unknown type, and that unknown type’s identity can be determined at run time by analyzing α . In λ_R , as in most other type systems, $\exists\alpha.\alpha$ implements an abstract type (in this particular example, a useless one), because no representation of α is provided. Dynamic types are implemented in λ_R by including a representation of the unknown type, as in $\exists\alpha.R(\alpha) \times \alpha$.

1.1 Expressiveness

In the interest of clarity of presentation, we express λ_R as an extension of Harper and Morrisett’s λ_i^{ML} and focus on their differences. The principal difference is the restriction of type analysis to those types for which representations are provided. This change does not diminish the expressiveness of our calculus; λ_i^{ML} may be translated in a straightforward syntax-directed manner into λ_R , as described in Section 4.

Moreover, we incorporate into the λ_R calculus an additional improvement in expressiveness over λ_i^{ML} that is independent of explicit type passing: In λ_i^{ML} , information gained by analyzing a type is not propagated to other variables having that type. Consequently, when analyzing a type α with the interest of processing an object of type α , it is necessary to create a function with argument type α and then apply that function to the object of interest. In other words, the type system of λ_i^{ML} requires the use of beta-expansions that are not operationally necessary. In λ_R we resolve this shortcoming by strengthening the typing rule for typecase so that it refines types in place. This strengthening is not intrinsic to λ_R , and an analogous rule could be added to λ_i^{ML} to the same benefit.

1.2 Overview

The remainder of this paper is organized as follows: In Section 2 we review the λ_i^{ML} calculus. We then present, in Section 3, our λ_R calculus and discuss its formal semantics, including representation terms, R -types, and the strengthened typecase rule. As examples of its expressiveness, in Section 4 we give an embedding of λ_i^{ML} in λ_R , and in Section 5, we discuss the simplification of polymorphic closure conversion by explicit type passing. We end with discussion of related work and conclusions in Sections 6 and 7. In the appendices we relate our typed semantics to an untyped one through type erasure (Appendix A), and provide the formal static semantics (Appendix B).

2 Intensional Type Analysis

Suppose we wanted to store efficiently an array of boolean values. Most computer architectures require that memory accesses are a word at a time, but it is a waste of space to store booleans as integers. A solution is to pack thirty-two booleans into one word and use bit manipulations to retrieve the correct value. To subscript from a packed boolean array, we might use the following function (with \ll for shift left, $\&$ for bitwise and, and \llcorner for inequality):

```
val bitsub : array[int] * int -> bool =
  fn (a,i) =>
    sub(a,i div 32) & (1<<(i mod 32)) <> 0
```

This function is fine when we know a given array contains boolean values, but we would like code polymorphic over all arrays to be able to use this mechanism. Below we define a new array constructor, `PackedArray`, which will produce an array of integers to hold booleans, and an ordinary array for other types. We also define an associated subscript operation, `packedsub`, which calls `bitsub` on arrays of booleans and the ordinary subscript operator on arrays of other types. These constructs can be implemented with intensional type analysis, where in both cases an argument type is examined with a “typecase” construct:

```
type PackedArray[ $\alpha$ ] =
  Typecase  $\alpha$  of
    bool => array[int]
    | _ => array[ $\alpha$ ]

val packedsub :  $\forall\alpha$ . PackedArray[ $\alpha$ ] * int ->  $\alpha$  =
  Fn  $\alpha$  =>
    typecase  $\alpha$  of
      bool => bitsub
      | _ => sub
```

2.1 The λ_i^{ML} calculus

To formalize the tools of intensional type analysis, we begin by summarizing Harper and Morrisett’s λ_i^{ML} calculus (1995). The λ_i^{ML} calculus provides these tools in a form that is relatively simple, but already quite powerful.

The syntax of λ_i^{ML} , with some minor modifications, appears in Figure 1. The complete static semantics appears in Appendix B, though we will include relevant rules in this section. A small-step call-by-value operational semantics of λ_i^{ML} appears in Figures 2 and 3. We write \mapsto for evaluation steps that apply to both λ_i^{ML} and λ_R , and \mapsto_i for evaluation steps that apply only to λ_i^{ML} . We write $E[E'/X]$ for the capture-avoiding substitution of E' for X in E . In all cases, we consider alpha-equivalent expressions to be identical.

The backbone of λ_i^{ML} is a predicative variant of Girard’s F_ω (1972; 1971) in which the quantified type $\forall\alpha:\kappa.\sigma$ ranges only over type constructors and “small” types

(kinds)	$\kappa ::= \mathbf{Type} \mid \kappa_1 \rightarrow \kappa_2$
(con's)	$c ::= \alpha \mid \lambda\alpha:\kappa.c \mid c_1 c_2 \mid \hat{\mathbf{int}} \mid c_1 \hat{\rightarrow} c_2 \mid c_1 \hat{\times} c_2$ $\mid \mathbf{Typeprec} \ c(c_{\mathbf{int}}, c_{\rightarrow}, c_{\times})$
(types)	$\sigma ::= T(c) \mid \mathbf{int} \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall\alpha:\kappa.\sigma \mid \exists\alpha:\kappa.\sigma$
(terms)	$e ::= i \mid x \mid \lambda x:\sigma.e \mid \mathbf{fix} \ f:\sigma.v \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \mid \Lambda\alpha:\kappa.e \mid e[c]$ $\mid \mathbf{pack} \ e \ \mathbf{as} \ \exists\alpha:\kappa.\sigma_1 \ \mathbf{hiding} \ \sigma_2 \mid \mathbf{unpack} \ \langle\alpha, x\rangle = e_1 \ \mathbf{in} \ e_2$ $\mid \mathbf{typecase}[\alpha.\sigma] \ c \ \mathbf{of}$ $\quad \mathbf{int} \Rightarrow e_{\mathbf{int}}$ $\quad \beta \rightarrow \gamma \Rightarrow e_{\rightarrow}$ $\quad \beta \times \gamma \Rightarrow e_{\times}$
(values)	$v ::= i \mid \lambda x:\sigma.e \mid \mathbf{fix} \ x:\sigma.v \mid \langle v_1, v_2 \rangle \mid \Lambda\alpha:\kappa.e \mid \mathbf{pack} \ v \ \mathbf{as} \ \exists\alpha.\sigma_1 \ \mathbf{hiding} \ \sigma_2$

Fig. 1. Syntax of λ_i^{ML}

$(\lambda x:\sigma.e)v \mapsto e[v/x]$	$(\mathbf{fix} \ f:\sigma.v)v' \mapsto_i (v[\mathbf{fix} \ f:\sigma.v/f])v'$	
$\pi_1 \langle v_1, v_2 \rangle \mapsto v_1$	$\pi_2 \langle v_1, v_2 \rangle \mapsto v_2$	
$\mathbf{unpack} \ \langle\alpha, x\rangle = (\mathbf{pack} \ v \ \mathbf{as} \ \exists\beta.\sigma_1 \ \mathbf{hiding} \ \sigma_2) \ \mathbf{in} \ e_2 \mapsto e_2[\sigma_2/\alpha, v/x]$		
$\frac{e_1 \mapsto e'_1}{e_1 e_2 \mapsto e'_1 e_2}$	$\frac{e \mapsto e'}{ve \mapsto ve'}$	$\frac{e \mapsto e'}{e[c] \mapsto e'[c]}$
$\frac{e \mapsto e'}{\pi_i e \mapsto \pi_i e'}$	$\frac{e_1 \mapsto e'_1}{\langle e_1, e_2 \rangle \mapsto \langle e'_1, e_2 \rangle}$	$\frac{e \mapsto e'}{\langle v, e \rangle \mapsto \langle v, e' \rangle}$
<hr style="border-top: 1px solid black;"/>		
$\mathbf{pack} \ e \ \mathbf{as} \ \exists\beta.\sigma_1 \ \mathbf{hiding} \ \sigma_2 \mapsto \mathbf{pack} \ e' \ \mathbf{as} \ \exists\beta.\sigma_1 \ \mathbf{hiding} \ \sigma_2$		
<hr style="border-top: 1px solid black;"/>		
$\mathbf{unpack} \ \langle\alpha, x\rangle = e \ \mathbf{in} \ e_2 \mapsto \mathbf{unpack} \ \langle\alpha, x\rangle = e' \ \mathbf{in} \ e_2$		

Fig. 2. Operational Semantics for core language

(i.e., monotypes), which do not include the quantified types. An explicit injection $T(c)$ converts a type constructor into a type. For example, $T(\hat{\mathbf{int}})$ is equal to the type \mathbf{int} , and $T(c_1 \hat{\rightarrow} c_2)$ is equal to $T(c_1) \rightarrow T(c_2)$.

The type analysis operators are **Typeprec** and **typecase** at the constructor and term levels respectively. These operators, given an argument type c , dispatch to an appropriate branch based on whether c is $\hat{\mathbf{int}}$, a constructor for a function type or a product type. The $[\alpha.\sigma]$ annotation in a **typecase** term is used to make type checking syntax-directed, and indicates that when given a type argument c , the **typecase** is to return a value of type $\sigma[c/\alpha]$. When α does not appear free in σ we often omit it.

$$\begin{array}{c}
 (\Lambda\alpha:\kappa.e)[c] \mapsto_i e[c/\alpha] \quad (\mathbf{fix} \ f:\sigma.v)[c] \mapsto_i (v[\mathbf{fix} \ f:\sigma.v/f])[c] \\
 \hline
 \frac{c \text{ normalizes to } \hat{\mathbf{int}}}{\mathbf{typecase} \ c \ (e_{\mathbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\mathbf{int}}} \\
 \frac{c \text{ normalizes to } (c_1 \hat{\rightarrow} c_2)}{\mathbf{typecase} \ c \ (e_{\mathbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\rightarrow}[c_1/\beta, c_2/\gamma]} \\
 \frac{c \text{ normalizes to } (c_1 \hat{\times} c_2)}{\mathbf{typecase} \ c \ (e_{\mathbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\times}[c_1/\beta, c_2/\gamma]}
 \end{array}$$

 Fig. 3. Operational Semantics for type application and **typecase**

```

fix tostring : (∀α: Type. T(α) → string).
Λα: Type.
typecase[δ. T(δ) → string] α of
  int ⇒ int2string
  string ⇒ λobj: string. obj
  β →̂ γ ⇒
    λobj: T(β →̂ γ). "function"
  β ×̂ γ ⇒
    λobj: T(β ×̂ γ).
    "<" ^ (tostring[β](π1 obj)) ^ ", " ^ (tostring[γ](π2 obj)) ^ ">"
    
```

 Fig. 4. The function *tostring*

Occasionally, for brevity, we will write **typecase** terms as

$$\mathbf{typecase}[\alpha.\sigma] \ c \ (e_{\mathbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}).$$

As an example of the use of type analysis in λ_i^{ML} (with the addition of another base type, *string*), consider the function *tostring* in Figure 4. This function uses **typecase** to produce a string representation of a data object. For example, the call *tostring* [**int**] 3 returns the string “3”. As we cannot provide any information about the implementation of functions, we just return the word “function” when one is encountered, as in the call:

$$\mathit{tostring} \ [(\hat{\mathbf{int}} \hat{\rightarrow} \hat{\mathbf{int}}) \hat{\times} \hat{\mathbf{int}}] \ \langle \lambda x: \mathbf{int} . x + 1, 3 \rangle$$

which returns:

“⟨function, 3⟩”

When the argument to *tostring* is a product type, the function calls itself recursively. In this branch, the type variables β and γ are bound to the types of the first and second components of the tuple, so that the recursive call can be instantiated with the correct type.

Type checking λ_i^{ML} is based on the judgments in Figure 5, which define well-formedness of type constructors, types and terms, as well as equivalence of type constructors and types. In these judgments, Γ is a unified type and kind context:

an ordered, partial map from constructor variables (α, β, \dots) to kinds, and term variables (x, y, \dots) to types. As before, we use \vdash for rules of these judgments that apply to both λ_i^{ML} and λ_R , employing \vdash_i for rules specific to λ_i^{ML} .

<u>Judgment</u>	<u>Meaning</u>
$\Gamma \vdash c : \kappa$	c is a valid constructor of kind κ
$\Gamma \vdash c_1 = c_2 : \kappa$	c_1 and c_2 are equal constructors
$\Gamma \vdash \sigma$	σ is a valid type
$\Gamma \vdash \sigma_1 = \sigma_2$	σ_1 and σ_2 are equal types
$\Gamma \vdash e : \sigma$	e is a term of type σ

Fig. 5. Judgments of λ_i^{ML}

With this intuition, the typing rule for **typecase** is the natural one (but we will see that this rule is unnecessarily restrictive):

$$\begin{array}{c}
 \Gamma \vdash_i c : \mathbf{Type} \quad \Gamma, \delta : \mathbf{Type} \vdash_i \sigma \quad \Gamma \vdash_i e_{\mathbf{int}} : \sigma[\hat{\mathbf{int}}/\delta] \\
 \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type} \vdash_i e_{\rightarrow} : \sigma[(\beta \hat{\rightarrow} \gamma)/\delta] \\
 \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type} \vdash_i e_{\times} : \sigma[(\beta \hat{\times} \gamma)/\delta] \\
 \hline
 \Gamma \vdash_i \left(\begin{array}{l} \mathbf{typecase}[\delta, \sigma] c \text{ of} \\ \hat{\mathbf{int}} \Rightarrow e_{\mathbf{int}} \\ \beta \hat{\rightarrow} \gamma \Rightarrow e_{\rightarrow} \\ \beta \hat{\times} \gamma \Rightarrow e_{\times} \end{array} \right) : \sigma[c/\delta]
 \end{array}$$

Often, to compute the result type σ of a **typecase** expression the constructor-level **Typerec** on the argument α will be required. **Typerec** allows the creation of new types by similar intensional analysis. Several examples of its use appear in Harper and Morrisett (1995), including type-directed data layout, marshalling and unboxing.

While recursion in the term-level **typecase** is handled by **fix**, at the the constructor level there is no such mechanism. For this reason, **Typerec** is essentially a “fold” operation (or catamorphism) over inductively defined types. It provides primitive recursion by calling itself recursively on all of the components of the argument type. Also unlike **typecase**, where the branches explicitly bind arguments for the components of the type, the c_{\rightarrow} and c_{\times} branches of **Typerec** are constructor functions. For example, if the argument of a **Typerec** operation is $c_1 \hat{\times} c_2$, then that operation reduces to its c_{\times} branch (a constructor function of four arguments) applied to the components c_1 and c_2 , and to the result of recursively computing the **Typerec** operation on those components.

$$\begin{aligned}
 \mathbf{Typerec} (c_1 \hat{\times} c_2) (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) = \\
 c_{\times} c_1 c_2 \\
 (\mathbf{Typerec} c_1 (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times})) \\
 (\mathbf{Typerec} c_2 (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}))
 \end{aligned}$$

The kinding rule for **Typerec** is again the natural one. To compute a constructor

of kind κ , present a type argument and three branches returning κ constructors:

$$\frac{\begin{array}{l} \Gamma \vdash c : \mathbf{Type} \quad \Gamma \vdash c_{\mathbf{int}} : \kappa \\ \Gamma \vdash c_{\rightarrow} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\ \Gamma \vdash c_{\times} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \end{array}}{\Gamma \vdash \mathbf{Type}\mathbf{rec} \ c (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) : \kappa}$$

$$\begin{array}{l} \text{(types)} \quad \sigma ::= T(c) \mid \mathbf{int} \mid \sigma_1 \rightarrow \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall \alpha : \kappa. \sigma \mid \exists \alpha : \kappa. \sigma \mid R(c) \\ \\ \text{(terms)} \quad e ::= i \mid x \mid \lambda x : \sigma. e \mid \mathbf{fix} \ f : \sigma. v \mid e_1 e_2 \mid \langle e_1, e_2 \rangle \mid \pi_1 e \mid \pi_2 e \\ \quad \quad \quad \mid \Lambda \alpha : \kappa. v \mid e[c] \mid \mathbf{pack} \ e \ \mathbf{as} \ \exists \alpha : \kappa. \sigma_1 \ \mathbf{hiding} \ \sigma_2 \\ \quad \quad \quad \mid \mathbf{unpack} \ \langle \alpha, x \rangle = e_1 \ \mathbf{in} \ e_2 \\ \quad \quad \quad \mid \mathbf{R}_{\mathbf{int}} \mid \mathbf{R}_{\rightarrow}[c_1, c_2](e_1, e_2) \mid \mathbf{R}_{\times}[c_1, c_2](e_1, e_2) \\ \quad \quad \quad \mid \mathbf{typecase}[\alpha. \sigma] e \ \mathbf{of} \\ \quad \quad \quad \quad \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}} \\ \quad \quad \quad \quad \mathbf{R}_{\rightarrow}[\beta, \gamma](x, y) \Rightarrow e_{\rightarrow} \\ \quad \quad \quad \quad \mathbf{R}_{\times}[\beta, \gamma](x, y) \Rightarrow e_{\times} \\ \\ \text{(values)} \quad v ::= i \mid \lambda x : \sigma. e \mid \mathbf{fix} \ f : \sigma. v \mid \langle v_1, v_2 \rangle \\ \quad \quad \quad \mid \Lambda \alpha : \kappa. v \mid (\mathbf{fix} \ f : \sigma. v)[c_1] \dots [c_n] \mid \mathbf{pack} \ v \ \mathbf{as} \ \exists \alpha. \sigma_1 \ \mathbf{hiding} \ \sigma_2 \\ \quad \quad \quad \mid \mathbf{R}_{\mathbf{int}} \mid \mathbf{R}_{\rightarrow}[c_1, c_2](v_1, v_2) \mid \mathbf{R}_{\times}[c_1, c_2](v_1, v_2) \end{array}$$

Fig. 6. Syntax of λ_R

$$\begin{array}{l} (\Lambda \alpha : \kappa. v)[c] \mapsto_R (v[c/\alpha]) \\ \\ (\mathbf{fix} \ f : \sigma. v)[c_1] \dots [c_n] v' \mapsto_R (v[\mathbf{fix} \ f : \sigma. v/f])[c_1] \dots [c_n] v' \\ \\ \mathbf{typecase}[\delta. \sigma] \mathbf{R}_{\mathbf{int}} (e_{\mathbf{int}}, \beta \gamma xy. e_{\rightarrow}, \beta \gamma xy. e_{\times}) \mapsto_R e_{\mathbf{int}} \\ \mathbf{typecase}[\alpha. \sigma](\mathbf{R}_{\rightarrow}[c_1, c_2](v_1, v_2))(e_{\mathbf{int}}, \beta \gamma xy. e_{\rightarrow}, \beta \gamma xy. e_{\times}) \mapsto_R e_{\rightarrow}[c_1/\beta, c_2/\gamma, v_1/x, v_2/y] \\ \mathbf{typecase}[\alpha. \sigma](\mathbf{R}_{\times}[c_1, c_2](v_1, v_2))(e_{\mathbf{int}}, \beta \gamma xy. e_{\rightarrow}, \beta \gamma xy. e_{\times}) \mapsto_R e_{\times}[c_1/\beta, c_2/\gamma, v_1/x, v_2/y] \\ \\ \frac{e \mapsto_R e'}{\mathbf{typecase}[\delta. \sigma] e (e_{\mathbf{int}}, \beta \gamma xy. e_{\rightarrow}, \beta \gamma xy. e_{\times}) \mapsto_R \mathbf{typecase}[\delta. \sigma] e' (e_{\mathbf{int}}, \beta \gamma xy. e_{\rightarrow}, \beta \gamma xy. e_{\times})} \\ \\ \frac{e_1 \mapsto_R e'_1}{\mathbf{R}_{\rightarrow}[c_1, c_2](e_1, e_2) \mapsto_R \mathbf{R}_{\rightarrow}[c_1, c_2](e'_1, e_2)} \quad \frac{e \mapsto_R e'}{\mathbf{R}_{\rightarrow}[c_1, c_2](v, e) \mapsto_R \mathbf{R}_{\rightarrow}[c_1, c_2](v, e')} \\ \\ \frac{e_1 \mapsto_R e'_1}{\mathbf{R}_{\times}[c_1, c_2](e_1, e_2) \mapsto_R \mathbf{R}_{\times}[c_1, c_2](e'_1, e_2)} \quad \frac{e \mapsto_R e'}{\mathbf{R}_{\times}[c_1, c_2](v, e) \mapsto_R \mathbf{R}_{\times}[c_1, c_2](v, e')} \end{array}$$

Fig. 7. Operational Semantics for λ_R

3 The λ_R calculus

Figure 6 presents the syntax of λ_R , which we describe in detail in the following section. The features distinguishing λ_R from λ_i^{ML} are highlighted. The syntactic classes for kinds and constructors of λ_R are identical to those of λ_i^{ML} , and are accordingly omitted from the figure.

3.1 Term Representations of Types

The key feature we add to the term language of λ_R is the representations of types as terms, which remain when the types themselves are ultimately erased. The base type, \mathbf{int} , has a corresponding representation constant \mathbf{Rint} . Likewise, non-base types have representation constructors; for example, the type constructor $\mathbf{int} \hat{\rightarrow} \mathbf{int}$ is represented by the term $\mathbf{R}_{\rightarrow}[\mathbf{int}, \mathbf{int}](\mathbf{Rint}, \mathbf{Rint})$.

The argument to the term level **typecase** is a type representation, instead of an actual type. For example, if the argument is of the form $\mathbf{R}_{\rightarrow}[c_1, c_2](v_1, v_2)$, the arrow branch (e_{\rightarrow}) is taken. The type variables β and γ are still bound to c_1 and c_2 , the types that v_1 and v_2 represent. Because we need not only the component types but also their representations, x and y are bound to v_1 and v_2 . Hence, the operational semantics establishes the following rule for evaluating **typecases** over arrow types:

$$\begin{array}{l} \mathbf{typecase}[\delta.c](\mathbf{R}_{\rightarrow}[c_1, c_2](v_1, v_2))(e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) \\ \mapsto_R \\ e_{\rightarrow}[c_1/\beta, c_2/\gamma, v_1/x, v_2/y] \end{array}$$

The operational semantics for λ_R is given in Figures 2 and 7. Recall that evaluation steps applying to both λ_i^{ML} and λ_R are written with \mapsto . Evaluation steps applying only to λ_R are written with \mapsto_R .

The operational semantics is designed to permit a type erasure interpretation. In the main body of this paper, we give the semantics with types included (this makes programs more readable, and greatly eases the proof of type safety); but the semantics is designed so that programs behave in the same manner with types (and attendant machinery such as type abstractions and applications) removed. This erasure property is verifiable by inspection, and is formalized in Appendix A.

To achieve the desired erasure property, a number of changes are made from λ_i^{ML} . Aside from the most notable change, the use of type representations in **typecase** expressions, there are other minor changes as well. For example, λ_R imposes a value restriction on type abstractions. Without this restriction, a type abstraction (necessarily a value (Harper & Lillibridge, 1993)) could erase to a non-value, thereby defeating the erasure property. Similarly, if $(\mathbf{fix} f:\sigma.v)[c]$ stepped to $v[\mathbf{fix} f:\sigma.v/f][c]$, as in λ_i^{ML} , then when viewed under the lens of erasure, $(\mathbf{fix} f.v)$ would unroll for no reason, despite being a value. Consequently, $(\mathbf{fix} f:\sigma.v)[c_1] \cdots [c_n]$ is taken to be a value; the internal fix does not unroll until the recursive function is applied to an actual value.

To assign types to term representations of types, we have extended the types of λ_R to include the R construct, where the representation of a type τ is given

$$\begin{array}{c}
\frac{\Gamma \vdash_R \tau : \mathbf{Type}}{\Gamma \vdash_R R(\tau)} \\
\\
\frac{}{\Gamma \vdash_R \mathbf{R}_{\text{int}} : R(\hat{\text{int}})} \quad \frac{\Gamma \vdash_R e_1 : R(\tau_1) \quad \Gamma \vdash_R e_2 : R(\tau_2)}{\Gamma \vdash_R \mathbf{R}_{\rightarrow}[\tau_1, \tau_2](e_1, e_2) : R(\tau_1 \hat{\rightarrow} \tau_2)} \\
\\
\frac{\Gamma \vdash_R e_1 : R(\tau_1) \quad \Gamma \vdash e_2 : R(\tau_2)}{\Gamma \vdash_R \mathbf{R}_{\times}[\tau_1, \tau_2](e_1, e_2) : R(\tau_1 \hat{\times} \tau_2)}
\end{array}$$

Fig. 8. Formation rules for representation type and representation terms

the type $R(\tau)$. The formation rules for the type $R(\tau)$ and for the representation terms appear in Figure 8. For example, the formation rule for the representation of function types states that if the two subterms, e_1 and e_2 , are type representations of τ_1 and τ_2 , then $\mathbf{R}_{\rightarrow}[\tau_1, \tau_2](e_1, e_2)$ will be a representation of $\tau_1 \hat{\rightarrow} \tau_2$.

As an example of the use of λ_R , the *tostring* function from the previous section can be translated into λ_R by requiring it to take an additional term argument, x_α for the representation of the argument type:

```

fix tostring : ( $\forall \alpha : \mathbf{Type} . R(\alpha) \rightarrow T(\alpha) \rightarrow \mathbf{string}$ ).
 $\Lambda \alpha : \mathbf{Type} . \lambda x_\alpha : R(\alpha)$ .
  typecase[ $\delta . T(\delta) \rightarrow \mathbf{string}$ ]  $x_\alpha$  of
     $\mathbf{R}_{\text{int}} \Rightarrow \text{int2string}$ 
     $\mathbf{R}_{\text{string}} \Rightarrow \lambda \text{obj} : \mathbf{string} . \text{obj}$ 
     $\mathbf{R}_{\rightarrow}[\beta, \gamma](x, y) \Rightarrow$ 
       $\lambda \text{obj} : T(\beta \hat{\rightarrow} \gamma) . \text{"function"}$ 
     $\mathbf{R}_{\times}[\beta, \gamma](x, y) \Rightarrow$ 
       $\lambda \text{obj} : T(\beta \hat{\times} \gamma) .$ 
       $\text{"<" } \wedge \text{(tostring } [\beta] x (\pi_1 \text{ obj})) \wedge$ 
       $\text{" , " } \wedge \text{(tostring } [\gamma] y (\pi_2 \text{ obj})) \wedge \text{">"}$ 

```

The static semantics we have defined ensures that these R -types are singleton types; for each one there is exactly one value which inhabits it. This fact allows us to express constraints between types and their representations at a very fine level. For instance, in the *tostring* example, the representation argument must be the representation of the type of the object.

3.2 In-place Refinement of Types

The typing rules of λ_i^{ML} often force an inelegant use of **typecase**. In the *tostring* example in Section 2, and in its λ_R rendition above, we created closures in each of the branches of the **typecase**. It would be slightly more efficient and much more convenient, in this case, if we could lift the lambdas outside of the **typecase**, so that the branches of the **typecase** are not functions. This would allow the application to the type information and argument to be uncurried. Then, instead of a closure, each branch of the **typecase** would return a string. We could then write this function

$$\begin{array}{c}
\Gamma, \alpha: \mathbf{Type}, \Gamma' \vdash_R e : R(\alpha) \\
\Gamma, \Gamma' [\hat{\mathbf{int}}/\alpha] \vdash_R e_{\mathbf{int}}[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha, \hat{\mathbf{int}}/\delta] \\
\Gamma, \beta: \mathbf{Type}, \gamma: \mathbf{Type}, x: R(\beta), y: R(\gamma), \Gamma' [(\beta \rightarrow \gamma)/\alpha] \vdash_R \\
\quad e_{\rightarrow}[(\beta \rightarrow \gamma)/\alpha] : \sigma[(\beta \rightarrow \gamma)/\alpha, (\beta \rightarrow \gamma)/\delta] \\
\Gamma, \beta: \mathbf{Type}, \gamma: \mathbf{Type}, x: R(\beta), y: R(\gamma), \Gamma' [(\beta \hat{\times} \gamma)/\alpha] \vdash_R \\
\quad e_{\times}[(\beta \hat{\times} \gamma)/\alpha] : \sigma[(\beta \hat{\times} \gamma)/\alpha, (\beta \hat{\times} \gamma)/\delta] \\
\quad (\alpha, \beta, \gamma \notin \text{Dom}(\Gamma, \Gamma')) \\
\hline
\Gamma, \alpha: \mathbf{Type}, \Gamma' \vdash_R \mathbf{typecase}[\delta.\sigma] e (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\alpha/\delta]
\end{array}$$

Fig. 9. The Variable Refining **typecase** Rule

as:

```

fix tostring : ( $\forall \alpha: \mathbf{Type}. R(\alpha) \rightarrow T(\alpha) \rightarrow \mathbf{string}$ ).
   $\lambda \alpha: \mathbf{Type}. b\lambda x_{\alpha}: R(\alpha). \lambda obj: T(\alpha).$ 
  typecase[string]  $x_{\alpha}$  of
    Rint  $\Rightarrow$  int2string obj
    Rstring  $\Rightarrow$  obj
    R $\rightarrow$ [ $\beta, \gamma$ ]( $x, y$ )  $\Rightarrow$ 
      "function"
    R $\times$ [ $\beta, \gamma$ ]( $x, y$ )  $\Rightarrow$ 
      "<"  $\wedge$  (tostring [ $\beta$ ]  $x$  ( $\pi_1$  obj))  $\wedge$ 
      ", "  $\wedge$  (tostring [ $\gamma$ ]  $y$  ( $\pi_2$  obj))  $\wedge$  ">"

```

The reason we could not write this function in λ_i^{ML} is that it requires the type of *obj* to change based on which branch of the **typecase** is selected. In λ_i^{ML} , all that is known in the product branch is that *obj* has type $T(\alpha)$; it is not known that it has type $T((\beta \hat{\times} \gamma))$. In order to project from it in the recursive calls, the typing rules would have to update the type of *obj* to reflect the fact that we know that α is $\beta \hat{\times} \gamma$ in the product branch.

With the right enhancement to the static semantics this optimization is possible. We have held off discussion of the λ_R 's **typecase** typing rule in order to emphasize this point. The basic idea is that in some cases **typecase** increases our knowledge of the argument type, and we can propagate this knowledge back to the type system. In the rule for type checking a **typecase** term, when the argument has type $R(\alpha)$, we refine all types containing α to reflect the gain in information. This refinement is done using a simple substitution, as shown in Figure 9.

For example, to typecheck the e_{\rightarrow} branch, we substitute $\beta \hat{\rightarrow} \gamma$ for α everywhere, including the surrounding context.² Consequently, the types of the variables bound in the context will be refined by that substitution. In contrast, in λ_i^{ML} this substitution is only made in the return type of each branch—not in the context—so in order to propagate the desired information one must abstract over all variables of interest.

² The substitution for α is applied within the branches themselves in order to avoid creating a hole in the scope of α . In practice, a typechecker would implement this operation by a local type definition, rather than by substitution.

$$\begin{array}{c}
\Gamma \vdash_R e : R(c) \quad \Gamma \vdash_R e_{\text{int}} : \sigma[\widehat{\text{int}}/\delta] \\
\Gamma, \beta: \mathbf{Type}, \gamma: \mathbf{Type}, x:R(\beta), y:R(\gamma) \vdash_R e_{\rightarrow} : \sigma[(\beta \dot{\rightarrow} \gamma)/\delta] \\
\Gamma, \beta: \mathbf{Type}, \gamma: \mathbf{Type}, x:R(\beta), y:R(\gamma) \vdash_R e_{\times} : \sigma[(\beta \dot{\times} \gamma)/\delta] \\
\hline
\Gamma \vdash_R \mathbf{typecase}[\delta.\sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[c/\delta] \quad (\beta, \gamma \notin \text{Dom}(\Gamma, \Gamma'))
\end{array}$$

Fig. 10. Non-refining **typecase** rule

Sometimes refinement is not possible even with this rule; such cases arise when the type being analyzed is not a variable.³ For such cases, our type system includes an ordinary non-refining typing rule as well (Figure 10).

3.3 Semantics

The static semantics of λ_R consists of a collection of rules for deriving judgments of the forms shown in Figure 5. The formal operational and static semantics of λ_R appear in Figure 7 and Appendix B, and from them we can prove several useful properties about λ_R .

Theorem 3.1 (Decidability)

It is decidable whether or not $\Gamma \vdash e : \tau$ is derivable in λ_R .

The proof of decidability of λ_R typechecking is merely an extension of the decidability of λ_i^{ML} typechecking to a few new constructs; full details of that proof appear in Morrisett (1995). This proof consists of two parts: showing that constructors and types may be reduced to a normal form, and showing that type derivations can be normalized to an equivalent syntax-directed version.

Next, we would like to show that the static semantics guarantees safety; that is, if a term typechecks, then the operational semantics will not get *stuck*. As usual, a term is considered stuck if it is not a value and no rule of our operational semantics applies to it.

Theorem 3.2 (Type Safety)

If $\emptyset \vdash e : \sigma$ and $e \mapsto^* e'$ then e' is not stuck.

Type safety is proved syntactically, in the manner popularized by Wright and Felleisen (Wright & Felleisen, 1994), employing the usual Progress and Subject Reduction Lemmas.

Lemma 3.3 (Progress)

If $\emptyset \vdash e : \tau$ and e is not a value then there exists an e' such that $e \mapsto e'$.

Lemma 3.4 (Subject Reduction)

If $\emptyset \vdash e : \tau$ and $e \mapsto e'$ then $\emptyset \vdash e' : \tau$.

³ Some non-variable cases can still be refined by the trivialization rules of the next section. Cases in which refinement is impossible are those in which the outermost type constructor cannot be determined statically, that is, irreducible application and **Typerec** expressions.

The proof of these lemmas is largely standard, following the pattern in Morrisett (1995), for example. However, one subtlety does arise as a result of in-place refinement. This subtlety arises in one of the usual substitution lemmas used in the Subject Reduction lemma:

Lemma 3.5 (Constructor Substitution into Terms)

If $\Gamma, \alpha : \kappa, \Gamma' \vdash e : \tau$ and $\emptyset \vdash c : \kappa$ then $\Gamma, \Gamma'[c/\alpha] \vdash e[c/\alpha] : \tau[c/\alpha]$.

In Lemma 3.5, suppose e is a **typecase** expression in which the type being analyzed is the variable of substitution α , and suppose the refining rule is used to typecheck e :

$$\frac{\begin{array}{c} \vdots \\ \Gamma, \alpha : \mathbf{Type}, \Gamma' \vdash e' : R(\alpha) \end{array} \quad \frac{\begin{array}{c} \vdots \\ \Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e_{\mathbf{int}}[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha, \hat{\mathbf{int}}/\delta] \end{array} \quad \dots}{\Gamma, \alpha : \mathbf{Type}, \Gamma' \vdash \mathbf{typecase}[\delta.\sigma] e' (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\alpha/\delta]}$$

After substitution for α , the type being analyzed is no longer a variable and the refining rule no longer applies. Not surprisingly, the non-refining rule does not generally suffice to typecheck these cases.

We resolve this problem by adding three new rules for typechecking analyses of types whose outermost constructor is known:

$$\frac{\Gamma \vdash e : R(\hat{\mathbf{int}}) \quad \Gamma \vdash e_{\mathbf{int}} : \sigma[\hat{\mathbf{int}}/\delta]}{\Gamma \vdash \mathbf{typecase}[\delta.\sigma] e (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\hat{\mathbf{int}}/\delta]}$$

$$\frac{\Gamma \vdash e : R(c_1 \rightarrow c_2) \quad \Gamma, x : R(c_1), y : R(c_2) \vdash e_{\rightarrow}[c_1/\beta, c_2/\gamma] : \sigma[(c_1 \dot{\rightarrow} c_2)/\delta]}{\Gamma \vdash \mathbf{typecase}[\delta.\sigma] e (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[(c_1 \dot{\rightarrow} c_2)/\delta]}$$

$$\frac{\Gamma \vdash e : R(c_1 \hat{\times} c_2) \quad \Gamma, x : R(c_1), y : R(c_2) \vdash e_{\times}[c_1/\beta, c_2/\gamma] : \sigma[(c_1 \hat{\times} c_2)/\delta]}{\Gamma \vdash \mathbf{typecase}[\delta.\sigma] e (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[(c_1 \hat{\times} c_2)/\delta]}$$

We call these *trivialization* rules because they typecheck trivial analyses, ones that can be eliminated statically. They operate by typechecking the relevant branch and discarding the remaining branches as dead code.

For example, suppose the substitutend c is $\hat{\mathbf{int}}$. Then we obtain:

$$\frac{\frac{\text{(by substitution)}}{\Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e'[\hat{\mathbf{int}}/\alpha] : R(\hat{\mathbf{int}})} \quad \frac{\text{(retained)}}{\Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e_{\mathbf{int}}[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha][\hat{\mathbf{int}}/\delta]}}{\Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash (\mathbf{typecase}[\delta.\sigma] e' (e_{\mathbf{int}}, \dots))[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha][\hat{\mathbf{int}}/\delta]}$$

and observe that $\sigma[\hat{\mathbf{int}}/\alpha][\hat{\mathbf{int}}/\delta] = \sigma[\alpha/\delta][\hat{\mathbf{int}}/\alpha]$, as desired.

In general, since c kindchecks in the empty context, it is easy to show that c is equivalent to a constructor whose outermost constructor is known. It follows that one of the three trivialization rules must apply, since if (for example) $c = c_1 \hat{\times} c_2$, then $R(c) = R(c_1 \times c_2)$, and consequently $e : R(c)$ implies $e : R(c_1 \times c_2)$.

<i>types</i>	$ T(c) = T(c)$	
	$ \mathbf{int} = \mathbf{int}$	
	$ \sigma_1 \rightarrow \sigma_2 = \sigma_1 \rightarrow \sigma_2 $	
	$ \sigma_1 \times \sigma_2 = \sigma_1 \times \sigma_2 $	
	$ \forall\alpha:\kappa.\sigma = \forall\alpha:\kappa.R(\alpha:\kappa) \rightarrow \sigma $	
	$ \exists\alpha:\kappa.\sigma = \exists\alpha:\kappa.R(\alpha:\kappa) \times \sigma $	
<i>expressions</i>		
	$ x = x$	
	$ i = i$	
	$ \lambda x:\sigma.e = \lambda x: \sigma . e $	
	$ \mathbf{fix} f:\sigma.v = \mathbf{fix} f: \sigma . v $	
	$ e_1 e_2 = e_1 e_2 $	
	$ \langle e_1, e_2 \rangle = \langle e_1 , e_2 \rangle$	
	$ \pi_1 e = \pi_1 e $	
	$ \pi_2 e = \pi_2 e $	
	$ \Lambda\alpha:\kappa.e = \Lambda\alpha:\kappa.\lambda x_\alpha:R(\alpha:\kappa). e $	
	$ e[c] = e [c] \mathbf{Rep}(c)$	
	$ \mathbf{pack} e \text{ as } (\exists\alpha:\kappa.\sigma_1) \mathbf{hiding} \sigma_2 = \mathbf{pack} \langle \mathbf{Rep}(\sigma_2), e \rangle$	
		$\text{as } \exists\alpha:\kappa.R(\alpha:\kappa) \times \sigma_1 \mathbf{hiding} \sigma_2 $
	$ \mathbf{unpack} \langle \alpha, x \rangle = e_1 \text{ in } e_2 = \mathbf{unpack} \langle \alpha, y \rangle = e_1 $	
		$\text{in } e_2 [\pi_1 y/x_\alpha, \pi_2 y/x]$
	$\left \begin{array}{l} \mathbf{typecase}[\alpha.\sigma] c \text{ of} \\ \mathbf{int} \Rightarrow e_{\mathbf{int}} \\ \beta \rightarrow \gamma \Rightarrow e_{\rightarrow} \\ \beta \times \gamma \Rightarrow e_{\times} \end{array} \right = \mathbf{typecase}[\alpha. \sigma] \mathbf{Rep}(c) \text{ of}$	
		$\mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}} $
		$\mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow e_{\rightarrow} $
		$\mathbf{R}_{\times}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow e_{\times} $

Fig. 11. Translation from λ_i^{ML} to λ_R

4 Embedding of λ_i^{ML}

We next describe an embedding of λ_i^{ML} expressions into λ_R . We include this embedding for two reasons: first, to show that λ_R is as expressive as λ_i^{ML} , and second, to demonstrate a simple use of λ_R as an intermediate language. The full details of the embedding appear in Figures 11, 12, and 13. The embedding of λ_i^{ML} types and terms is written $|\sigma|$ and $|e|$.

The main difference between λ_i^{ML} and λ_R is the **typecase** term; in λ_i^{ML} it takes a type constructor as its argument, in λ_R it takes a term representing a type. Therefore, to simulate a λ_i^{ML} **typecase** term with an λ_R **typecase** term, we need to be able to form the term representation of the type constructor argument. This operation, written $\mathbf{Rep}(\cdot)$, appears in Figure 12.

Creating the representation of given type constructor is complicated by the fact that the argument to **Typerec** can (and often will) contain constructors with free type variables. These type variables are translated to term variables that represent them, but in order to do this translation, we must maintain the invariant that for every accessible type variable, a corresponding term variable representing it is also accessible. We make this guarantee by a process reminiscent of phase splitting (Harper *et al.*, 1990) or evidence passing (Jones, 1992). In the translation of constructor abstractions, we split the abstractions to take both the constructor and a

$$\begin{aligned}
\text{Rep}(\hat{\text{int}}) &= \mathbf{R}_{\text{int}} \\
\text{Rep}(\tau_1 \hat{\rightarrow} \tau_2) &= \mathbf{R}_{\rightarrow}[\tau_1, \tau_2](\text{Rep}(\tau_1), \text{Rep}(\tau_2)) \\
\text{Rep}(\tau_1 \hat{\times} \tau_2) &= \mathbf{R}_{\times}[\tau_1, \tau_2](\text{Rep}(\tau_1), \text{Rep}(\tau_2)) \\
\text{Rep}(\alpha) &= x_\alpha \\
\text{Rep}(\lambda\alpha:\kappa.c) &= \Lambda\alpha:\kappa.\lambda x_\alpha:R(\alpha:\kappa).\text{Rep}(c) \\
\text{Rep}(c_1 c_2) &= \text{Rep}(c_1)[c_2]\text{Rep}(c_2) \\
\text{Rep}(\mathbf{Typerec} \tau(c_{\text{int}}, c_{\rightarrow}, c_{\times})) &= (\mathbf{fix} f:\forall\alpha:\mathbf{Type}.R(\alpha) \rightarrow R(c^*[\alpha]:\kappa). \\
&\quad \Lambda\alpha:\mathbf{Type}. \\
&\quad \lambda x_\alpha:R(\alpha). \\
&\quad \mathbf{typecase}[R(c^*[\alpha]:\kappa)] x_\alpha \\
&\quad \quad \mathbf{R}_{\text{int}} \Rightarrow \text{Rep}(c_{\text{int}}) \\
&\quad \quad \mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
&\quad \quad \quad \text{Rep}(c_{\rightarrow})[\beta]x_\beta [\gamma]x_\gamma \\
&\quad \quad \quad [c^*[\beta]](f[\beta]x_\beta) [c^*[\gamma]](f[\gamma]x_\gamma) \\
&\quad \quad \mathbf{R}_{\times}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
&\quad \quad \quad \text{Rep}(c_{\times})[\beta]x_\beta [\gamma]x_\gamma \\
&\quad \quad \quad [c^*[\beta]](f[\beta]x_\beta) [c^*[\gamma]](f[\gamma]x_\gamma) \\
&\quad) [\tau] \text{Rep}(\tau) \\
&\quad \text{where } c^*[\tau] = \mathbf{Typerec} \tau'(c_{\text{int}}, c_{\rightarrow}, c_{\times}) \\
&\quad \text{and } \kappa \text{ is the kind of the full } \mathbf{Typerec} \text{ expression}
\end{aligned}$$

Fig. 12. Translation of constructors to their representations

$$\begin{aligned}
R(\tau : \mathbf{Type}) &\stackrel{\text{def}}{=} R(\tau) \\
R(c : \kappa_1 \rightarrow \kappa_2) &\stackrel{\text{def}}{=} \forall\alpha:\kappa_1. R(\alpha : \kappa_1) \rightarrow R(c\alpha : \kappa_2)
\end{aligned}$$

Fig. 13. Representations of higher constructors

term variable, where the term variable is constrained to be the representation of that constructor, and application is also changed accordingly:

$$\begin{aligned}
|\Lambda\alpha:\kappa.e| &= \Lambda\alpha:\kappa.\lambda x_\alpha:R(\alpha:\kappa).|e| \\
|e[c]| &= |e| [c] \text{Rep}(c)
\end{aligned}$$

Note that this translation also satisfies the value restriction placed on λ_R type abstractions. Dually, we also include the representation of a type constructor when we form an existential package. The notation $R(\alpha:\kappa)$ is defined shortly.

The next issue to address is the representation of higher-order type constructors. If, for example, c has kind $\mathbf{Type} \rightarrow \mathbf{Type}$, it maps type arguments to type results. Accordingly, the representation of c maps the representation of c 's type argument to the representation of c 's type result. More generally, when c has kind $\kappa_1 \rightarrow \kappa_2$, its representation is a polymorphic function taking the representation of c 's constructor argument to the representation of the result of applying c to that argument. When c has kind κ , we define $R(c : \kappa)$ to be the type of c 's representation, as given in Figure 13.

The last issue in our translation of type constructors to their representations is the definition of the representation of a $\mathbf{Typerec}$ constructor. We represent it as

$$\begin{aligned}
\text{Spl}(\cdot) &= \cdot \\
\text{Spl}(\Gamma, x:\tau) &= \text{Spl}(\Gamma), x:\tau \\
\text{Spl}(\Gamma, \alpha:\kappa) &= \text{Spl}(\Gamma), \alpha:\kappa, x_\alpha:R(\alpha:\kappa)
\end{aligned}$$

Fig. 14. Context splitting

a **typecase** on the representation of the argument to the **Typerec**, but because **Typerec** is recursive, we must wrap the **typecase** in a recursive polymorphic function:

$$\begin{aligned}
\text{Rep}(\mathbf{Typerec} \tau(c_{\text{int}}, c_{\rightarrow}, c_{\times})) &= ((\mathbf{fix} f : \forall \alpha: \mathbf{Type} . R(\alpha) \rightarrow R(c^*[\alpha]:\kappa). \\
&\quad \Lambda \alpha: \mathbf{Type} . \lambda x_\alpha: R(\alpha). \\
&\quad \mathbf{typecase} x_\alpha \\
&\quad \quad R_{\text{int}} \Rightarrow \text{Rep}(c_{\text{int}}) \\
&\quad \quad \dots) \\
&[\tau] \text{Rep}(\tau))
\end{aligned}$$

where $c^*[\tau] = \mathbf{Typerec} \tau(c_{\text{int}}, c_{\rightarrow}, c_{\times})$ and κ is the result kind of the **Typerec**.

In the arrow and product of the **typecase**, this function must be called recursively on the subcomponents of the type, just as in **Typerec**. For example, consider the arrow case:

$$\begin{aligned}
\mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) &\Rightarrow \\
\text{Rep}(c_{\rightarrow}) \quad [\beta] x_\beta \quad [\gamma] x_\gamma \quad [c^*[\beta]] (f[\beta] x_\beta) \quad [c^*[\gamma]] (f[\gamma] x_\gamma)
\end{aligned}$$

The c_{\rightarrow} arm of the **Typerec** is a function taking four type variables, the first two being β and γ , the second two being the results of calling the **Typerec** recursively on β and γ . However, because of phase splitting in the translation, each type argument has an associated term argument for its representation, so the translation of c_{\rightarrow} , takes four pairs of type and term arguments. For the first two pairs, β and γ , their representations x_β and x_γ are readily available from the **typecase**. For the recursive arguments, we use the original **Typerec** to find the resulting constructors and call f recursively to find the representations of those resulting constructors.

4.1 Correctness of the Embedding

The static and dynamic correctness of the embedding is not difficult to show. In what follows, we write \vdash_i for typing derivations in λ_i^{ML} and \vdash_R for typing derivations in λ_R .

We begin by establishing a lemma, stating that the representations defined above have the appropriate type. Recall that the definition of representations required an inductive assumption that representations are always available for constructor variables. This invariant is enforced using an auxiliary definition to split contexts (written $\text{Spl}(\Gamma)$), explicitly adding representations for each variable in the context.

Lemma 4.1

If $\Gamma \vdash_R c : \kappa$ then $\text{Spl}(\Gamma) \vdash_R \text{Rep}(c) : R(c : \kappa)$

Now we can establish the static correctness of the embedding:

Theorem 4.2 (Static correctness)

Define $|\Gamma|$ as $\text{Spl}(\Gamma')$, where Γ' is defined as the pointwise translation of Γ (that is, for all $x \in \Gamma$, $\Gamma'(x) = |\Gamma(x)|$, and for all $\alpha \in \Gamma$, $\Gamma'(\alpha) = \Gamma(\alpha)$). Then:

1. If $\Gamma \vdash_i c : \kappa$ then $|\Gamma| \vdash_R c : \kappa$
2. If $\Gamma \vdash_i c_1 = c_2 : \kappa$ then $|\Gamma| \vdash_R c_1 = c_2 : \kappa$
3. If $\Gamma \vdash_i \sigma$ then $|\Gamma| \vdash_R |\sigma|$
4. If $\Gamma \vdash_i \sigma_1 = \sigma_2$ then $|\Gamma| \vdash_R |\sigma_1| = |\sigma_2|$
5. If $\Gamma \vdash_i e : \tau$ then $|\Gamma| \vdash_R |e| : |\tau|$

$$\begin{aligned}
\text{Rep}[\hat{\mathbf{int}}] &= \{\mathbf{R}_{\mathbf{int}}\} \\
\text{Rep}[\tau_1 \hat{\rightarrow} \tau_2] &= \{\mathbf{R}_{\rightarrow}[\tau'_1, \tau'_2](e_1, e_2) \mid \Gamma \vdash \tau_i = \tau'_i : \mathbf{Type}, e_i \in \text{Rep}[\tau_i], i = 1, 2\} \\
\text{Rep}[\tau_1 \hat{\times} \tau_2] &= \{\mathbf{R}_{\times}[\tau'_1, \tau'_2](e_1, e_2) \mid \Gamma \vdash \tau_i = \tau'_i : \mathbf{Type}, e_i \in \text{Rep}[\tau_i], i = 1, 2\} \\
\text{Rep}[\alpha] &= \{x_\alpha\} \\
\text{Rep}[\lambda\alpha:\kappa.\hat{c}] &= \{\Lambda\alpha:\kappa.\lambda x_\alpha:R(\alpha:\kappa).e \mid e \in \text{Rep}[\hat{c}]\} \\
\text{Rep}[\hat{c}_1 \hat{c}_2] &= \{e_1[\hat{c}'_2]e_2 \mid e_1 \in \text{Rep}[\hat{c}_1], \Gamma \vdash c_2 = c'_2 : \kappa, e_2 \in \text{Rep}[\hat{c}_2]\} \\
\text{Rep}[\mathbf{Typerec} \tau \quad] &= \{(\mathbf{fix} f:\forall\alpha:\mathbf{Type}.R(\alpha) \rightarrow R(c^*[\alpha]:\kappa). \\
(c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) &\quad \Lambda\alpha:\mathbf{Type}.\lambda x_\alpha:R(\alpha). \\
&\quad \mathbf{typecase}[R(c^*[\alpha]:\kappa)] x_\alpha \\
&\quad \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}} \\
&\quad \mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
&\quad \quad e_{\rightarrow}[\beta]x_\beta [\gamma]x_\gamma \\
&\quad \quad [c^*[\beta]](f[\beta]x_\beta) [c^*[\gamma]](f[\gamma]x_\gamma) \\
&\quad \mathbf{R}_{\times}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
&\quad \quad e_{\times}[\beta]x_\beta [\gamma]x_\gamma \\
&\quad \quad [c^*[\beta]](f[\beta]x_\beta) [c^*[\gamma]](f[\gamma]x_\gamma) \\
&\quad) [\tau'] e \mid e_{\mathbf{int}} \in \text{Rep}[\hat{c}_{\mathbf{int}}], e_{\rightarrow} \in \text{Rep}[\hat{c}_{\rightarrow}], e_{\times} \in \text{Rep}[\hat{c}_{\times}], \\
&\quad \Gamma \vdash \tau' = \tau : \mathbf{Type}, e \in \text{Rep}[\tau] \} \\
&\text{where } c^*[\tau'] = \mathbf{Typerec} \tau'(c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) \\
&\text{and } \kappa \text{ is the kind of the full } \mathbf{Typerec} \text{ expression}
\end{aligned}$$

$$\overline{\text{Rep}[c]} = \{e \mid \Gamma \vdash c = c' : \kappa \ \& \ e \in \text{Rep}[c']\}$$

Fig. 15. Extended representations

In order to show the dynamic correctness of the embedding, we must show that the result of translation simulates the operation of λ_i^{ML} . However, because the evaluation of the term representations does not exactly match the reduction of constructors, we must add some imprecision to the simulation. We allow constructors and their representations appearing in the result of the embedding to be of any equivalent constructor (based on the definition of constructor equality), instead of exactly matching the constructor appearing in the source λ_i^{ML} term.

We define the operation $\text{Rep}[c]$ which produces a set of representations of the constructor c , in Figure 15. For any c , $\text{Rep}(c)$ is in the set $\text{Rep}[c]$. The other members of this set differ from $\text{Rep}(c)$ only in the embedded constructors. For example, $\text{Rep}[\hat{\mathbf{int}} \hat{\rightarrow} \hat{\mathbf{int}}]$ includes both $\mathbf{R}_{\rightarrow}[\hat{\mathbf{int}}, \hat{\mathbf{int}}](\mathbf{R}_{\mathbf{int}}, \mathbf{R}_{\mathbf{int}})$, and

types

$$\begin{aligned}
\llbracket T(c) \rrbracket &= \{T(c') \mid \Gamma \vdash c = c' : \mathbf{Type}\} \\
\llbracket \mathbf{int} \rrbracket &= \{\mathbf{int}\} \\
\llbracket \sigma_1 \rightarrow \sigma_2 \rrbracket &= \{\sigma'_1 \rightarrow \sigma'_2 \mid \sigma_i \in \llbracket \sigma_i \rrbracket, i = 1, 2\} \\
\llbracket \sigma_1 \times \sigma_2 \rrbracket &= \{\sigma'_1 \times \sigma'_2 \mid \sigma_i \in \llbracket \sigma_i \rrbracket, i = 1, 2\} \\
\llbracket \forall \alpha : \kappa. \sigma \rrbracket &= \{\forall \alpha : \kappa. R(\alpha : \kappa) \rightarrow \sigma' \mid \sigma' \in \llbracket \sigma \rrbracket\} \\
\llbracket \exists \alpha : \kappa. \sigma \rrbracket &= \{\exists \alpha : \kappa. R(\alpha : \kappa) \times \sigma' \mid \sigma' \in \llbracket \sigma \rrbracket\}
\end{aligned}$$

expressions

$$\begin{aligned}
\llbracket x \rrbracket &= \{x\} \\
\llbracket i \rrbracket &= \{i\} \\
\llbracket \lambda x : \sigma. e \rrbracket &= \{\lambda x : \sigma'. e' \mid \sigma' \in \llbracket \sigma \rrbracket, e' \in \llbracket e \rrbracket\} \\
\llbracket \mathbf{fix} f : \sigma. v \rrbracket &= \{\mathbf{fix} f : \sigma'. v' \mid \sigma' \in \llbracket \sigma \rrbracket, v' \in \llbracket v \rrbracket\} \\
\llbracket e_1 e_2 \rrbracket &= \{e'_1 e'_2 \mid e'_i \in \llbracket e_i \rrbracket, i = 1, 2\} \\
\llbracket \langle e_1, e_2 \rangle \rrbracket &= \{\langle e'_1, e'_2 \rangle \mid e'_i \in \llbracket e_i \rrbracket, i = 1, 2\} \\
\llbracket \pi_1 e \rrbracket &= \{\pi_1 e' \mid e' \in \llbracket e \rrbracket\} \\
\llbracket \pi_2 e \rrbracket &= \{\pi_2 e' \mid e' \in \llbracket e \rrbracket\} \\
\llbracket \Lambda \alpha : \kappa. e \rrbracket &= \{\Lambda \alpha : \kappa. \lambda x_\alpha : R(\alpha : \kappa). e' \mid e' \in \llbracket e \rrbracket\} \\
\llbracket e[c] \rrbracket &= \{e'[c'] e'' \mid e' \in \llbracket e \rrbracket, \Gamma \vdash c = c' : \kappa, e'' \in \overline{\text{Rep}[c]}\} \\
\llbracket \left[\begin{array}{l} \mathbf{pack} e \text{ as } (\exists \alpha : \kappa. \sigma) \\ \mathbf{hiding} c \end{array} \right] \rrbracket &= \left\{ \left[\begin{array}{l} \mathbf{pack} \langle e'', e' \rangle \\ \mathbf{as} \exists \alpha : \kappa. R(\alpha : \kappa) \times \sigma' \\ \mathbf{hiding} c' \end{array} \right] \mid \begin{array}{l} e'' \in \overline{\text{Rep}[c]} \\ e' \in \llbracket e \rrbracket, \sigma' \in \llbracket \sigma \rrbracket \\ \Gamma \vdash c' = c : \kappa \end{array} \right\} \\
\llbracket \mathbf{unpack} \langle \alpha, x \rangle = e_1 \text{ in } e_2 \rrbracket &= \left\{ \left[\begin{array}{l} \mathbf{unpack} \langle \alpha, y \rangle = e'_1 \\ \mathbf{in} (\lambda x_\alpha : R(\alpha : \kappa). \lambda x : \alpha. e'_2)(\pi_1 y)(\pi_2 y) \end{array} \right] \mid \begin{array}{l} e'_1 \in \llbracket e_1 \rrbracket \\ e'_2 \in \llbracket e_2 \rrbracket \end{array} \right\} \\
\llbracket \left[\begin{array}{l} \mathbf{typecase}[\alpha. \sigma] c \text{ of} \\ \mathbf{int} \Rightarrow e_{\mathbf{int}} \\ \beta \hat{\rightarrow} \gamma \Rightarrow e_{\rightarrow} \\ \beta \times \gamma \Rightarrow e_{\times} \end{array} \right] \rrbracket &= \left\{ \left[\begin{array}{l} \mathbf{typecase}[\alpha. \sigma'] e \text{ of} \\ \mathbf{R}_{\mathbf{int}} \Rightarrow e'_{\mathbf{int}} \\ \mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow e'_{\rightarrow} \\ \mathbf{R}_{\times}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow e'_{\times} \end{array} \right] \mid \begin{array}{l} e \in \overline{\text{Rep}[c]} \\ e'_{\mathbf{int}} \in \llbracket e_{\mathbf{int}} \rrbracket \\ e'_{\rightarrow} \in \llbracket e_{\rightarrow} \rrbracket \\ e'_{\times} \in \llbracket e_{\times} \rrbracket \\ \sigma' \in \llbracket \sigma \rrbracket \end{array} \right\}
\end{aligned}$$

Fig. 16. Extended translation

$\mathbf{R}_{\rightarrow}[(\lambda \beta : \mathbf{Type}. \beta) \hat{\mathbf{int}}, \hat{\mathbf{int}}](\mathbf{R}_{\mathbf{int}}, \mathbf{R}_{\mathbf{int}})$. The set $\overline{\text{Rep}[c]}$, defined at the bottom of the figure, is even larger. It includes all representations of equivalent constructors. For example, not only does $\overline{\text{Rep}[\hat{\mathbf{int}} \hat{\rightarrow} \hat{\mathbf{int}}]}$ include the above terms, but it also includes a representation of $((\lambda \beta : \mathbf{Type}. \beta) \mathbf{int}) \rightarrow \mathbf{int}$

$$\mathbf{R}_{\rightarrow}[(\lambda \beta : \mathbf{Type}. \beta) \hat{\mathbf{int}}, \hat{\mathbf{int}}](\Lambda \beta : \mathbf{Type}. \lambda x_\beta : R(\beta). x) \mathbf{R}_{\mathbf{int}}, \mathbf{R}_{\mathbf{int}}).$$

Likewise, the operations $\llbracket \sigma \rrbracket$ and $\llbracket e \rrbracket$ in Figure 16 generalize the translation of λ_i^{ML} types and terms. Again $|\sigma|$ is in the set $\llbracket \sigma \rrbracket$ and $|e|$ is in $\llbracket e \rrbracket$. In these sets, embedded constructors and their representations may be replaced with equivalent forms. For example, $\llbracket T(\hat{\mathbf{int}}) \rrbracket$ includes both the types $T(\hat{\mathbf{int}})$ and $T((\lambda \beta : \mathbf{Type}. \beta) \hat{\mathbf{int}})$. For the translation of terms, $\llbracket x[\hat{\mathbf{int}}] \rrbracket$ includes $x[\hat{\mathbf{int}}] \mathbf{R}_{\mathbf{int}}$, $x[(\lambda \beta : \mathbf{Type}. \beta) \hat{\mathbf{int}}] \mathbf{R}_{\mathbf{int}}$, and $x[\hat{\mathbf{int}}](\Lambda \beta : \mathbf{Type}. \lambda x_\beta : R(\beta). x) \mathbf{R}_{\mathbf{int}}$.

To begin, we must establish how substitution interacts with these operations. In

the following, we will use the following abbreviations (where S_1 and S_2 are arbitrary sets of terms):

$$\begin{aligned} S_1[S_2/x] &\stackrel{\text{def}}{=} \{e[e'/x] \mid e \in S_1 \ \& \ e' \in S_2\} \\ S_1[e'/x] &\stackrel{\text{def}}{=} S_1[\{e'\}/x] \end{aligned}$$

Lemma 4.3 (Substitution)

1. If $\Gamma, \alpha:\kappa \vdash c' : \kappa'$ and $\Gamma \vdash c : \kappa$, then $\text{Rep}[c'] [c/\alpha] [\overline{\text{Rep}[c]}/x_\alpha] \subseteq \overline{\text{Rep}[c' [c/\alpha]]}$.
2. If $\Gamma, \alpha:\kappa \vdash c' : \kappa'$ and $\Gamma \vdash c : \kappa$, then $\text{Rep}[c'] [c/\alpha] [\overline{\text{Rep}[c]}/x_\alpha] \subseteq \overline{\text{Rep}[c' [c/\alpha]]}$.
3. If $\Gamma, \alpha:\kappa \vdash c' : \kappa'$ and $\Gamma \vdash c : \kappa$, then $\overline{\text{Rep}[c'] [c/\alpha] [\overline{\text{Rep}[c]}/x_\alpha]} \subseteq \overline{\text{Rep}[c' [c/\alpha]]}$.
4. If $\Gamma, \alpha:\kappa \vdash_i \sigma$ and $\Gamma \vdash c : \kappa$ then $\llbracket \sigma \rrbracket [c/\alpha] \subseteq \llbracket \sigma [c/\alpha] \rrbracket$.
5. If $\Gamma, \alpha:\kappa \vdash_i e : \sigma$ and $\Gamma \vdash c' = c : \kappa$, then $\llbracket e \rrbracket [c'/\alpha] [\overline{\text{Rep}[c']}/x_\alpha] \subseteq \llbracket e [c/\alpha] \rrbracket$.
6. If $\Gamma, x:\sigma \vdash_i e : \sigma'$ and $\Gamma \vdash_i v : \sigma$ then $\llbracket e \rrbracket [\llbracket v \rrbracket /x] = \llbracket e [v/x] \rrbracket$.

Next, we also need to establish that the evaluation of term representations agrees with constructor equality. In the end, our goal is to show that if $e \in \overline{\text{Rep}[\mathbf{int}]}$ then e must evaluate to $\mathbf{R}_{\mathbf{int}}$ (and similar results for arrow and product types).

Lemma 4.4

For all $\emptyset \vdash c : \kappa$, $e \in \text{Rep}[c]$ then either e is a value or there exists some e' and c' such that $e \mapsto^+ e'$ and $e' \in \text{Rep}[c']$ and c reduces to c' .

Lemma 4.5

If $e \in \overline{\text{Rep}[c]}$ then e evaluates to a value $v \in \overline{\text{Rep}[c]}$.

Corollary 4.6

1. If $e \in \overline{\text{Rep}[\mathbf{int}]}$ then e evaluates to $\mathbf{R}_{\mathbf{int}}$.
2. If $e \in \overline{\text{Rep}[\tau_1 \hat{\rightarrow} \tau_2]}$ then e evaluates to $\mathbf{R}_{\rightarrow}[\tau'_1, \tau'_2](v_1, v_2)$, where $\emptyset \vdash \tau_i = \tau'_i : \mathbf{Type}$ and $v_i \in \overline{\text{Rep}[\tau_i]}$ for $i = 1, 2$.
3. If $e \in \overline{\text{Rep}[\tau_1 \times \tau_2]}$ then e evaluates to $\mathbf{R}_{\times}[\tau'_1, \tau'_2](v_1, v_2)$, where $\emptyset \vdash \tau_i = \tau'_i : \mathbf{Type}$ and $v_i \in \overline{\text{Rep}[\tau_i]}$ for $i = 1, 2$.

Lemma 4.7 (Simulation)

If $\vdash_i e_1 : \sigma$ and $e_1 \mapsto_i e_2$ then for all $e'_1 \in \llbracket e_1 \rrbracket$ there exists an $e'_2 \in \llbracket e_2 \rrbracket$ such that $e_1 \mapsto_R^* e'_2$.

Now we can conclude the dynamic correctness of the translation:

Theorem 4.8 (Dynamic Correctness)

If $\vdash e : \mathbf{int}$ and $e \mapsto_i^* i$ then $|e| \mapsto_R^* i$.

5 Typed Closure Conversion

As a final example, we consider typed closure conversion in an impredicative, λ_R -like framework. Our analysis will show that typed closure conversion is much simpler in our setting, and will shed light on which mechanisms from typed closure conversion in the type-passing setting are actually essential.

The goal of closure conversion is to eliminate nested lambdas and produce an equivalent program where all functions are defined only at the top level. This is done

by replacing all inner functions with explicit closures that are represented within the language as pairs consisting of a function pointer (the code of the closure), and a tuple (the environment of the closure). The environment contains values for the free variables of the function. The function pointer is bound globally to a function that abstracts the environment as well as the arguments of the function and is thus closed. Application is rewritten so that the code of a closure is first applied to its environment and then to its arguments.

The development in this section is given at an informal level, as fully formalizing the type theory in our discussion would take us too far from our key points. Formalizations of the type theory necessary to this section are given for type erasure in Crary and Weirich (1999), and for type passing in Minamide *et al.* (1996).

5.1 Monomorphic Typed Closure Conversion

The challenge of typed closure conversion is to preserve the typing properties of the program. If two source expressions have the same source type, they should have the same target type. Consider first the typed closure conversion of a monomorphic language (Minamide *et al.*, 1996). In order that functions having the same type but different free variables will have equivalent types after closure conversion, an existential type is used to hold the type of the environment abstract. Therefore, a function of type $\tau_1 \rightarrow \tau_2$ is translated to a closure of type $\exists\alpha.((\tau_1 \times \alpha) \rightarrow \tau_2) \times \alpha$. For example, to closure convert the following function declaration (containing the free variables $x : \mathbf{int}$ and $y : \mathbf{bool}$),

$$\mathbf{val} \text{ mymonofunc} = \lambda f : \mathbf{int} \rightarrow \mathbf{int} . \langle f x, y \rangle$$

we need to abstract over the free variables x and y . This changes the lambda to expect two arguments, the first being the argument f and the second being an environment consisting of a tuple containing the values for the free variables x and y .

$$\lambda z : (\mathbf{int} \rightarrow \mathbf{int}) \times (\mathbf{int} \times \mathbf{bool}) . \langle (\pi_1 z)(\pi_1(\pi_2 z)), (\pi_2(\pi_2 z)) \rangle$$

To simplify the examples, we will use pattern matching syntax in lambdas, and write this function as:

$$\lambda \langle f : \mathbf{int} \rightarrow \mathbf{int}, \langle x : \mathbf{int}, y : \mathbf{bool} \rangle \rangle . \langle f x, y \rangle$$

Since the argument f in the source term is a function, it too must be closure converted. Therefore f is taken to have the type:

$$\sigma_f = \exists\alpha.(\mathbf{int} \times \alpha \rightarrow \mathbf{int}) \times \alpha$$

In order to apply f , our main function must unpack it, extract the code pointer and its environment, and then apply the code to both x and that environment:

$$\lambda \langle f : \sigma_f, \langle x : \mathbf{int}, y : \mathbf{bool} \rangle \rangle . \\ \langle \mathbf{unpack} \langle \alpha, f' : (\mathbf{int} \times \alpha \rightarrow \mathbf{int}) \times \alpha \rangle = f \mathbf{in} (\pi_1 f') \langle x, \pi_2 f' \rangle, y \rangle$$

This lambda abstraction is closed and may be hoisted to the top level. Suppose this hoisting is performed and the closed lambda is given the name `mymonocode`.

It remains to construct the closure for `mymonofunc`, by pairing the code pointer (`mymonocode`) with its environment, and then hiding the environment’s type in an existential package:

```
val mymonoclosure = pack ⟨mymonocode, ⟨x, y⟩⟩ as ∃α.(σf × α → int) × α
  hiding int × bool
```

5.2 Polymorphic Typed Closure Conversion

In the monomorphic case there is no discrepancy between type-passing (Minamide *et al.*, 1996) and type-erasure (Morrisett *et al.*, 1999) closure conversion. However, with the introduction of polymorphism, significant differences arise. The differences stem from the fact that functions may contain free type variables as well as free value variables, and closed code must abstract both. This abstraction of code over free type variables is performed in the same manner in both settings; the differences arise in when the closure is constructed.

In a type-erasure semantics, where type application has no run-time effect, it is possible to resolve the code’s abstracted type variables when the closure is created, simply by applying the code to the appropriate type arguments. In principle, this would mean performing the indicated type substitution at run time (an unacceptable run-time cost), but since types are erased this need not take place in reality; the instantiated “duplicate” is no different from the original and may share with it in memory.

In a type-passing semantics, types are real run-time data so this strategy is impermissible. Instead, free type variables are collected into an environment in the same manner as free value variables. Operationally, this is dealt with in exactly the same manner as for value variables (as discussed above); however, considerably more type-theoretic machinery is required in the target language in order to typecheck the resulting closure (Minamide *et al.*, 1996).

For example, consider the following function declaration (containing the free type variables α and β and the free value variables $y : \beta$ and $z : \mathbf{int}$, for some appropriately typed term $e_{x,y,z}$):

```
val myfunc : α → (int × β) = λx:α. ex,y,z
```

The closed version of this function abstracts over the free type and value variables:

$$\Lambda\gamma: \mathbf{Type} \times \mathbf{Type}. \lambda z: (\pi_1\gamma \times (\pi_2\gamma \times \mathbf{int})). e_{(\pi_1z), (\pi_1(\pi_2z)), (\pi_2(\pi_2z))}$$

In pattern matching notation:

$$\Lambda\langle\alpha: \mathbf{Type}, \beta: \mathbf{Type}\rangle. \lambda\langle x: \alpha, \langle y: \beta, z: \mathbf{int}\rangle\rangle. e_{x,y,z}$$

Suppose that this closed code is hoisted to the top level and given the name `mycode`. Observe that `mycode` has the type:

$$\forall\gamma: \kappa_{\text{tenv}}. (\pi_1\gamma \times \tau_{\text{venv}}) \rightarrow (\mathbf{int} \times \pi_2\gamma)$$

where $\kappa_{\text{tenv}} = \mathbf{Type} \times \mathbf{Type}$ and $\tau_{\text{venv}} = \pi_2\gamma \times \mathbf{int}$. It remains to build a closure from `mycode`.

5.2.1 Type-passing Closures

In the example above, observe that although the function is intended to take an argument of type α , the type of `mycode` indicates an argument of type $\pi_1\gamma$, where γ is the type environment. Thus, if `mycode` is applied to the “wrong” type argument (one for which the first component is not α), the code cannot be used as intended. However, nothing prevents `mycode` from being applied to any constructor having kind κ_{tenv} . Therefore, the first step to building a closure is to constrain `mycode` to be applied only to the appropriate type argument:

$$\text{val myclosure}_1 : \forall \gamma : \kappa_{tenv} = \langle \alpha, \beta \rangle. (\alpha \times \tau_{tenv}) \rightarrow (\mathbf{int} \times \beta) = \text{mycode}$$

This type uses a translucent type (Harper & Lillibridge, 1994) to dictate that the constructor argument γ must be $\langle \alpha, \beta \rangle$. This type constraint can be understood in two steps: first, a subtyping step to add the constraint, and, second, an equality step (since it follows from this constraint that $\pi_1\gamma = \alpha$ and $\pi_2\gamma = \beta$),

$$\begin{aligned} & \forall \gamma : \kappa_{tenv}. (\pi_1\gamma \times \tau_{tenv}) \rightarrow (\mathbf{int} \times \pi_2\gamma) \\ & \leq \forall \gamma : \kappa_{tenv} = \langle \alpha, \beta \rangle. (\pi_1\gamma \times \tau_{tenv}) \rightarrow (\mathbf{int} \times \pi_2\gamma) \\ & = \forall \gamma : \kappa_{tenv} = \langle \alpha, \beta \rangle. (\alpha \times \sigma_{tenv}) \rightarrow (\mathbf{int} \times \beta) \end{aligned}$$

where $\sigma_{tenv} = \beta \times \mathbf{int} \cong \tau_{tenv}[\langle \alpha, \beta \rangle / \gamma]$. The need for this translucency mechanism and the type theory supporting it are described in much greater detail in Minamide, *et al.* (1996).

The next step in constructing a closure from `mycode` is to pair `myclosure_1` with the value environment $(\langle y, z \rangle)$ and the type environment $(\langle \alpha, \beta \rangle)$. The operator for pairing values with types is existential packaging, so we obtain:

$$\text{val myclosure}_2 = \text{pack } \langle \text{myclosure}_1, \langle y, z \rangle \rangle \text{ as } \sigma_{\text{myclosure}_2} \text{ hiding } \langle \alpha, \beta \rangle$$

with type:

$$\begin{aligned} & \sigma_{\text{myclosure}_2} = \\ & \exists \delta_{tenv} : \kappa_{tenv}. (\forall \gamma : \kappa_{tenv} = \delta_{tenv}. (\alpha \times \sigma_{tenv}) \rightarrow (\mathbf{int} \times \beta)) \times \sigma_{tenv} \end{aligned}$$

The final step is to hide (as before) the type of the value environment, and (unlike before) the *kind* of the type environment, obtaining:

$$\begin{aligned} \text{val myclosure} = & \text{packkind} \\ & \text{pack myclosure}_2 \text{ as } \dots \text{ hiding } \sigma_{tenv} \\ & \text{as } \sigma_{\text{myclosure}} \text{ hiding } \kappa_{tenv} \end{aligned}$$

with type:

$$\begin{aligned} & \sigma_{\text{myclosure}} = \\ & \exists \kappa_{tenv} : \mathbf{Kind}. \exists \epsilon_{tenv} : \mathbf{Type}. \exists \delta_{tenv} : \kappa_{tenv}. \\ & (\forall \gamma : \kappa_{tenv} = \delta_{tenv}. (\alpha \times \epsilon_{tenv}) \rightarrow (\mathbf{int} \times \beta)) \times \epsilon_{tenv} \end{aligned}$$

More generally, a closure for a function of type $\tau_1 \rightarrow \tau_2$ will have the type:⁴

$$\begin{aligned} & \exists k_{tenv} : \mathbf{Kind} . \exists \epsilon_{venv} : \mathbf{Type} . \exists \delta_{tenv} : k_{tenv} . \\ & (\forall \gamma : k_{tenv} = \delta_{tenv} . (\tau_1 \times \epsilon_{venv}) \rightarrow \tau_2) \times \epsilon_{venv} \end{aligned}$$

This illustrates that building a closure in a type-passing setting requires two heavy-weight type-theoretic constructs: first, a translucent type mechanism, so that the code may be constrained to be applied only to the correct type environment, and, second, a special form of existential type for abstracting kinds.⁵

5.2.2 Type-erasure Closures

In a type-erasure setting, things work out more simply than in the type-passing setting. Since type application has no run-time effect, the closed code can simply be applied to its type environment when the closure is constructed; there is no need to defer that application by including the type environment in the closure. Also, no explicit translucency mechanism is required to ensure that the correct type environment is used, since the code is eagerly applied to the correct type environment at the outset.

The simplest account of typed closure conversion in a type-erasure setting is given in Morrisett, *et al.* (1999), but that account does not support intensional type analysis, so it is not entirely comparable to the type-passing account of Minamide, *et al.* (1996) summarized above. For comparable expressive power, the function requires representations of the free type variables α and β so that its body can analyze these types. Once we add these representations to the context, however, we may proceed using exactly the closure conversion process of Morrisett, *et al.*, and it is instructive to observe what happens.

For example, `myfunc` is rewritten to

$$\mathbf{val\ myfunc}' : \alpha \rightarrow (\mathbf{int} \times \beta) = \lambda x : \alpha . e_{x,y,z,w_\alpha,w_\beta}$$

containing the additional free value variables $w_\alpha : R(\alpha)$ and $w_\beta : R(\beta)$.

The closed version of this function (in pattern matching notation) is:

$$\Lambda \langle \alpha : \mathbf{Type}, \beta : \mathbf{Type} \rangle . \lambda \langle x : \alpha, \langle y : \beta, \langle z : \mathbf{int}, \langle w_\alpha : R(\alpha), w_\beta : R(\beta) \rangle \rangle \rangle \rangle . e_{x,y,z,w_\alpha,w_\beta}$$

Again, suppose this code is hoisted and given the name `mycode'`. Then `mycode'` has type:

$$\forall \gamma : \kappa_{tenv} . (\pi_1 \gamma \times \tau'_{venv}) \rightarrow (\mathbf{int} \times \pi_2 \gamma)$$

where $\tau'_{venv} = \pi_2 \gamma \times (\mathbf{int} \times (R(\pi_1 \gamma) \times R(\pi_2 \gamma)))$.

To build a closure from `mycode'`, we first apply it to the appropriate type environment:

$$\mathbf{val\ myclosure_1}' : (\alpha \times \sigma'_{venv}) \rightarrow (\mathbf{int} \times \beta) = \mathbf{mycode}'[(\alpha, \beta)]$$

⁴ Except that any function types within τ_1 and τ_2 must be closure-converted themselves.

⁵ By an abuse of terminology, this second mechanism is often called an “existential kind”, even though the existential itself is a type.

where $\sigma'_{\text{env}} = \beta \times (\mathbf{int} \times (R(\alpha) \times R(\beta))) \cong \tau'_{\text{env}}[\langle \alpha, \beta \rangle / \gamma]$. Next we pair `myclosure_1'` with its value environment, obtaining:

$$\mathbf{val\ myclosure_2'} = \langle \mathbf{myclosure_1'}, \langle x, \langle y, \langle w_\alpha, w_\beta \rangle \rangle \rangle \rangle$$

with type $((\alpha \times \sigma'_{\text{env}}) \rightarrow (\mathbf{int} \times \beta)) \times \sigma'_{\text{env}}$.

Finally, we hide the type of the value environment, obtaining:

$$\mathbf{val\ myclosure'} = \mathbf{pack\ myclosure_2'} \mathbf{as\ } \sigma'_{\text{myclosure}} \mathbf{hiding\ } \sigma'_{\text{env}}$$

with type:

$$\sigma'_{\text{myclosure}} = \exists \delta. ((\alpha \times \delta) \rightarrow (\mathbf{int} \times \beta)) \times \delta$$

More generally, a closure for a function of type $\tau_1 \rightarrow \tau_2$ will have the type:

$$\exists \delta. ((\tau_1 \times \delta) \rightarrow \tau_2) \times \delta$$

Observe that, despite the support for intensional type analysis, like Morrisett, *et al.* (1999) we are able to give the same type for closures as in the monomorphic case.

Consider what remains of the key mechanisms of type-passing closure conversion, translucent types and abstract kinds: The type for `myclosure_1'` dictates that its environment contains representations for two types, and specifically (via the R -type mechanism) for the types α and β in particular. In other words, the R -type mechanism provides a sense of translucency; the code cannot be applied to (environments containing) representations of arbitrary types, just the particular indicated types. Thus, although we avoid the full-blown translucency mechanism of Minamide *et al.* (1996), some form of translucency nevertheless emerges as essential.

On the other hand, the need for abstract kinds, like the *particular* form of translucency used by Minamide *et al.*, appears to be a requirement of type-passing closure conversion only. In the quasi-type-passing system that we propose here, it is replaced by a standard existential over constructors, thereby simplifying the type theory.

6 Related Work

Closely related to our work is the work of Minamide on lifting of type parameters for tag-free garbage collection (Minamide, 1997). Minamide was interested in lifting type parameters out of code so they could be preallocated at compile time. His lifting procedure required the maintenance of interrelated constraints between type parameters to retain type soundness, and he used a system similar to ours that makes explicit the passing of type parameters in order to simplify the expression of such constraints. The principal difference between Minamide's system and ours is that Minamide did not consider intensional type analysis. Minamide's system also makes a distinction between type representations (which he calls *evidence*, following Jones (1992)) and ordinary terms, while λ_R type representations are fully first-class. Finally, his system does not support separate compilation well while a transformation to λ_R can be applied uniformly across modules.

The issue of type parameter lifting is an important one for compilers based on λ_R . The construction of type representations at run time would likely lead to significant cost and, in practice, should be lifted out to compile time whenever possible. (Unfortunately, in the presence of polymorphic recursion, which λ_R supports, it is not always possible.) Mechanisms for such lifting have been developed by Minamide (in the work discussed above) and by Saha and Shao (1998).

Dubois *et al.* (1995) also pass explicit type representations to polymorphic functions when compiling ad-hoc polymorphism. However, their system differs from ours and Minamide’s in that no mechanism is provided for connecting representations to the types they denote, and consequently, information gained by analyzing type representations does not propagate into the type system.

Duggan (1998) proposes another typed framework for intensional type analysis that is similar in some ways to λ_i^{ML} . Like λ_i^{ML} , Duggan’s system passes types implicitly and allows for the intensional analysis of types at the term level. Duggan’s system does not support intensional type analysis at the constructor level, as λ_i^{ML} and λ_R do, but it adds a facility for defining type classes (using union and recursive kinds) and allows type analysis to be restricted to members of such classes.

Since the results of this paper were first announced (Crary *et al.*, 1998), work has continued on the topic of intensional type analysis in type-erasure settings: Crary and Weirich (1999) proposed a somewhat involved but highly expressive type theory in which the mechanisms of this paper can be used as a simple programming idiom, instead of as primitive language mechanisms. This type theory has the advantage that the source-level type structure can be preserved for the purpose of intensional type analysis even through program transformation that change types. This makes the type theory compatible with low-level typed intermediate languages in a type-preserving compiler, resolving a proposed direction for future work from our original report. Furthermore, Crary and Weirich show that their type theory supports the intensional analysis of polymorphic types, which λ_i^{ML} and λ_R do not.

Also, the mechanisms of this paper were used by Saha, *et al.* (2000) to develop a type-erasure-compatible version of the type system of Trifonov, *et al.* (2000). Trifonov, *et al.*’s type system extends λ_i^{ML} with kind polymorphism, thereby allowing the analysis of polymorphic types represented with higher-order abstract syntax. Because the kind of the bound variable is held abstract, they may analyze polymorphic types with quantifiers ranging over higher kinds. In contrast, Crary and Weirich’s mechanism limits quantifiers to the base kind and uses first-order analysis.

7 Conclusions and Future Directions

We have presented a type-theoretic framework that supports the passing and analysis of type information at run time, but that avoids the shortcomings associated with previous such frameworks (*e.g.*, duplication of constructs, lack of abstraction, and complication of closure conversion). This new framework makes it feasible to use intensional type analysis in settings where the shortcomings previously made it impractical.

For example, Morrisett *et al.* (1999) developed typing mechanisms for low-level

intermediate and target languages that allow type information to be used all the way to the end of compilation. It would be desirable, in a system based on those mechanisms, to be able to exploit that type information using intensional type analysis. Unfortunately, the shortcomings of type-passing semantics made it incompatible with some of those low-level typing mechanisms. This unfortunate incompatibility has made it infeasible to use the mechanisms of Morrisett *et al.* in type-analyzing compilers such as TIL/ML (Tarditi *et al.*, 1996; Morrisett *et al.*, 1996) and FLINT (Shao, 1997b), and has made it infeasible to use intensional type analysis in the end-to-end typed compiler TALC (Morrisett *et al.*, 1999). The framework in this paper makes it possible to unify these two lines of work for the first time.

Another important question is whether a parametricity theorem like that of Reynolds (1983) can be shown for λ_R . Polymorphism is clearly non-parametric in λ_i^{ML} , but the lowering of type analysis to explicit term-level representatives makes it plausible that some sort of parametricity could be shown for λ_R . In other words, we discussed at an intuitive level in Section 1 how the explicit passing of types restores the ability to abstract types that was discarded by λ_i^{ML} ; it would be interesting to explore how that intuition may be formalized.

References

- Abadi, Martín, Banerjee, Anindya, Heintze, Nevin, & Riecke, Jon G. 1999 (Jan.). A core calculus of dependency. *Pages 147–160 of: Twenty-sixth ACM SIGACT-SIGPLAN symposium on principles of programming languages.*
- Aditya, Shail, & Caro, Alejandro. 1993 (June). Compiler-directed type reconstruction for polymorphic languages. *Pages 74–82 of: Conference on functional programming languages and computer architecture.*
- Barendregt, H. P. (1992). Lambda calculi with types. Abramsky, S., Gabbay, Dov M., & Maibaum, T. S. E. (eds), *Handbook of logic in computer science*, vol. 2. Oxford: Oxford Science Publishers.
- Birkedal, Lars, Tofte, Mads, & Vejlstrup, Magnus. 1996 (Jan.). From region inference to von Neumann machines via region representation inference. *Pages 171–183 of: Twenty-third ACM SIGACT-SIGPLAN symposium on principles of programming languages.*
- Constable, Robert L. 1982 (June). *Intensional analysis of functions and types*. Tech. rept. CSR-118-82. Department of Computer Science, University of Edinburgh.
- Constable, Robert L., & Zlatin, Daniel R. (1984). The type theory of PL/CV3. *ACM transactions on programming languages and systems*, **6**(1), 94–117.
- Crary, Karl, & Weirich, Stephanie. 1999 (Sept.). Flexible type analysis. *Pages 233–248 of: 1999 ACM SIGPLAN international conference on functional programming.*
- Crary, Karl, Weirich, Stephanie, & Morrisett, Greg. 1998 (Sept.). Intensional polymorphism in type-erasure semantics. *Pages 301–312 of: 1998 ACM SIGPLAN international conference on functional programming.*
- Dubois, Catherine, Rouaix, François, & Weis, Pierre. 1995 (Jan.). Extensional polymorphism. *Pages 118–129 of: Twenty-second ACM SIGACT-SIGPLAN symposium on principles of programming languages.*
- Duggan, Dominic. 1998 (Mar.). A type-based semantics for user-defined marshalling in polymorphic languages. *Second workshop on types in compilation.*
- Girard, Jean-Yves. (1971). Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. *Pages*

- 63–92 of: Fenstad, J. E. (ed), *Proceedings of the second scandinavian logic symposium*. North-Holland Publishing Co.
- Girard, Jean-Yves. (1972). *Interprétation fonctionnelle et élimination des coupures de l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.
- Harper, Robert, & Lillibridge, Mark. 1993 (Jan.). Explicit polymorphism and CPS conversion. *Pages 206–219 of: Twentieth ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Harper, Robert, & Lillibridge, Mark. 1994 (Jan.). A type-theoretic approach to higher-order modules with sharing. *Pages 123–137 of: Twenty-first ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Harper, Robert, & Morrisett, Greg. 1995 (Jan.). Compiling polymorphism using intensional type analysis. *Pages 130–141 of: Twenty-second ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Harper, Robert, Mitchell, John C., & Moggi, Eugenio. 1990 (Jan.). Higher-order modules and the phase distinction. *Pages 341–354 of: Seventeenth ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Jones, Mark P. (1992). A theory of qualified types. *Fourth european symposium on programming*. Lecture Notes in Computer Science, vol. 582. Rennes, France: Springer-Verlag.
- Leroy, Xavier. (1992). Unboxed objects and polymorphic typing. *Pages 177–188 of: Nineteenth ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Minamide, Yasuhiko. (1997). *Full lifting of type parameters*. Submitted for publication. Earlier version published as “Compilation Based on a Calculus for Explicit Type-Passing” in the *Second Fuji International Workshop on Functional and Logic Programming*, 1996.
- Minamide, Yasuhiko, Morrisett, Greg, & Harper, Robert. 1996 (Jan.). Typed closure conversion. *Pages 271–283 of: Twenty-third ACM SIGACT-SIGPLAN symposium on principles of programming languages*.
- Mitchell, John C., & Plotkin, Gordon D. (1988). Abstract types have existential type. *ACM transactions on programming languages and systems*, **10**(3), 470–502.
- Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R., & Lee, P. 1996 (Feb.). The TIL/ML compiler: Performance and safety through types. *Workshop on compiler support for systems software*.
- Morrisett, Greg. 1995 (Dec.). *Compiling with types*. Ph.D. thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania. Published as Carnegie Mellon University School of Computer Science Technical Report CMU-CS-95-226.
- Morrisett, Greg, & Harper, Robert. (1997). Semantics of memory management for polymorphic languages. Gordon, A. D., & Pitts, A. M. (eds), *Higher order operational techniques in semantics*. Cambridge University Press.
- Morrisett, Greg, Felleisen, Matthias, & Harper, Robert. (1995). Abstract models of memory management. *Conference on functional programming languages and computer architecture*.
- Morrisett, Greg, Walker, David, Crary, Karl, & Glew, Neal. (1999). From System F to typed assembly language. *ACM transactions on programming languages and systems*, **21**(3), 527–568. An earlier version appeared in the 1998 Symposium on Principles of Programming Languages.
- Reynolds, John C. (1983). Types, abstraction and parametric polymorphism. *Pages 513–523 of: Information processing '83*. North-Holland. Proceedings of the IFIP 9th World Computer Congress.

- Ruf, Erik. 1997 (Jan.). Partitioning dataflow analyses using types. *Pages 15–26 of: Twenty-fourth ACM SIGACT-SIGPLAN symposium on principles of programming languages.*
- Saha, Bratin, & Shao, Zhong. 1998 (Mar.). Optimal type lifting. *Second workshop on types in compilation.*
- Saha, Bratin, Trifonov, Valery, & Shao, Zhong. 2000 (Sept.). Fully reflexive intensional type analysis in type erasure semantics. *Third workshop on types in compilation.*
- Shao, Zhong. 1997a (June). Flexible representation analysis. *Pages 85–98 of: 1997 ACM SIGPLAN international conference on functional programming.*
- Shao, Zhong. 1997b (June). An overview of the FLINT/ML compiler. *1997 workshop on types in compilation.* ACM SIGPLAN, Amsterdam. Published as Boston College Computer Science Department Technical Report BCCS-97-03.
- Tarditi, D., Morrisett, G., Cheng, P., Stone, C., Harper, R., & Lee, P. 1996 (May). TIL: A type-directed optimizing compiler for ML. *Pages 181–192 of: 1996 ACM SIGPLAN conference on programming language design and implementation.*
- Tolmach, Andrew. 1994 (June). Tag-free garbage collection using explicit type parameters. *Pages 1–11 of: ACM conference on Lisp and functional programming.*
- Trifonov, Valery, Saha, Bratin, & Shao, Zhong. 2000 (Sept.). Fully reflexive intensional type analysis. *Pages 82–93 of: Fifth ACM SIGPLAN international conference on functional programming.*
- Wright, Andrew K., & Felleisen, Matthias. (1994). A syntactic approach to type soundness. *Information and computation*, **115**, 38–94.

A Untyped Variant of λ_R

Although the formal static and operational semantics for λ_R are for a typed language, we would like to emphasize the point that types are unnecessary for computation and can safely be erased. Accordingly, we exhibit an untyped language, λ_R° , a translation of λ_R to this language through type erasure, and the following theorem, which states that execution in the untyped language mirrors execution in the typed language:

Theorem A.1

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

From this theorem and type safety for λ_R it follows that our untyped semantics is safe.

Corollary A.2

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

A.1 Syntax of Untyped Calculus

(terms) $u ::= i \mid x \mid \lambda x.u \mid \mathbf{fix} f.w \mid u_1 u_2$
 $\mid \langle u_1, u_2 \rangle \mid \pi_1 u \mid \pi_2 u \mid \mathbf{R}_{\mathbf{int}}$
 $\mid \mathbf{R}_{\rightarrow}(u_1, u_2) \mid \mathbf{R}_{\times}(u_1, u_2)$
 $\mid \mathbf{typecase} u \text{ of}$
 $\quad \mathbf{R}_{\mathbf{int}} \Rightarrow u_{\mathbf{int}}$
 $\quad \mathbf{R}_{\rightarrow}(x, y) \Rightarrow u_{\rightarrow}$
 $\quad \mathbf{R}_{\times}(x, y) \Rightarrow u_{\times}$

(values) $w ::= i \mid \lambda x.u \mid \mathbf{fix} f.w \mid \langle w_1, w_2 \rangle$
 $\mid \mathbf{R}_{\mathbf{int}} \mid \mathbf{R}_{\times}(w_1, w_2) \mid \mathbf{R}_{\rightarrow}(w_1, w_2)$

A.2 Type Erasure

$$\begin{aligned} x^\circ &= x \\ i^\circ &= i \\ \langle e_1, e_2 \rangle^\circ &= \langle e_1^\circ, e_2^\circ \rangle \\ (\pi_i e)^\circ &= \pi_i e^\circ \\ (\lambda x:c.e)^\circ &= \lambda x.e^\circ \\ (\Lambda \alpha:\kappa.v)^\circ &= v^\circ \\ (\mathbf{fix} f:c.v)^\circ &= \mathbf{fix} f.v^\circ \\ (e_1 e_2)^\circ &= e_1^\circ e_2^\circ \\ e[c]^\circ &= e^\circ \\ \mathbf{pack} e \text{ as } c \text{ hiding } c'^\circ &= e^\circ \\ \mathbf{unpack} \langle \alpha, x \rangle = e_1 \text{ in } e_2^\circ &= (\lambda x.e_2^\circ) e_1^\circ \\ \mathbf{R}_{\mathbf{int}}^\circ &= \mathbf{R}_{\mathbf{int}} \\ \mathbf{R}_{\rightarrow}[c_1, c_2](e_1, e_2)^\circ &= \mathbf{R}_{\rightarrow}(e_1^\circ, e_2^\circ) \\ \mathbf{R}_{\times}[c_1, c_2](e_1, e_2)^\circ &= \mathbf{R}_{\times}(e_1^\circ, e_2^\circ) \\ (\mathbf{typecase}[c] e \text{ of}) &= \mathbf{typecase} e^\circ \text{ of} \\ \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}} &= \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}}^\circ \\ \mathbf{R}_{\rightarrow}(x, y) \text{ as } (\beta \rightarrow \gamma) \Rightarrow e_{\rightarrow} &= \mathbf{R}_{\rightarrow}(x, y) \Rightarrow e_{\rightarrow}^\circ \\ \mathbf{R}_{\times}(x, y) \text{ as } (\beta \times \gamma) \Rightarrow e_{\times} &= \mathbf{R}_{\times}(x, y) \Rightarrow e_{\times}^\circ \end{aligned}$$

A.3 Operational Semantics of λ_R°

$$\begin{aligned} (\lambda x.u)w &\mapsto u[w/x] \\ (\mathbf{fix} f.w)w' &\mapsto (w[\mathbf{fix} f.w/f])w' \\ \pi_1 \langle w_1, w_2 \rangle &\mapsto w_1 \quad \pi_2 \langle w_1, w_2 \rangle \mapsto w_2 \\ \mathbf{typecase} \mathbf{R}_{\mathbf{int}}(u_{\mathbf{int}}, xy.u_{\rightarrow}, xy.u_{\times}) &\mapsto u_{\mathbf{int}} \end{aligned}$$

$$\mathbf{typecase} (\mathbf{R}_\times(w_1, w_2)) (u_{\mathbf{int}}, xy.u_\rightarrow, xy.u_\times) \mapsto u_\times[w_1, w_2/x, y]$$

$$\mathbf{typecase} (\mathbf{R}_\rightarrow(w_1, w_2)) (u_{\mathbf{int}}, xy.u_\rightarrow, xy.u_\times) \mapsto u_\rightarrow[w_1, w_2/x, y]$$

$$\frac{u_1 \mapsto u'_1}{u_1 u_2 \mapsto u'_1 u_2} \quad \frac{u \mapsto u'}{wu \mapsto wu'}$$

$$\frac{u_1 \mapsto u'_1}{\langle u_1, u_2 \rangle \mapsto \langle u'_1, u_2 \rangle} \quad \frac{u \mapsto u'}{\langle w, u \rangle \mapsto \langle w, u' \rangle}$$

$$\frac{u \mapsto u'}{\pi_1 u \mapsto \pi_1 u'} \quad \frac{u \mapsto u'}{\pi_2 u \mapsto \pi_2 u'}$$

$$\frac{u_1 \mapsto u'_1}{\mathbf{R}_\rightarrow(u_1, u_2) \mapsto \mathbf{R}_\rightarrow(u'_1, u_2)} \quad \frac{u \mapsto u'}{\mathbf{R}_\rightarrow(w, u) \mapsto \mathbf{R}_\rightarrow(w, u')}$$

$$\frac{u_1 \mapsto u'_1}{\mathbf{R}_\times(u_1, u_2) \mapsto \mathbf{R}_\times(u'_1, u_2)} \quad \frac{u \mapsto u'}{\mathbf{R}_\times(w, u) \mapsto \mathbf{R}_\times(w, u')}$$

$$\frac{u \mapsto u'}{\mathbf{typecase} u (u_{\mathbf{int}}, xy.u_\rightarrow, xy.u_\times) \mapsto \mathbf{typecase} u' (u_{\mathbf{int}}, xy.u_\rightarrow, xy.u_\times)}$$

B Static Semantics of λ_i^{ML} and λ_R

B.1 Constructor Formation

$\Gamma \vdash c : \kappa$

$$\frac{}{\Gamma \vdash \hat{\mathbf{int}} : \mathbf{Type}}$$

$$\frac{}{\Gamma \vdash \alpha : \kappa} \quad (\Gamma(\alpha) = \kappa)$$

$$\frac{\Gamma \vdash c_1 : \mathbf{Type} \quad \Gamma \vdash c_2 : \mathbf{Type}}{\Gamma \vdash c_1 \hat{\rightarrow} c_2 : \mathbf{Type}}$$

$$\frac{\Gamma \vdash c_1 : \mathbf{Type} \quad \Gamma \vdash c_2 : \mathbf{Type}}{\Gamma \vdash c_1 \hat{\times} c_2 : \mathbf{Type}}$$

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

$$\frac{\Gamma \vdash c_1 : \kappa_1 \rightarrow \kappa_2 \quad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1 c_2 : \kappa_2}$$

$$\frac{\begin{array}{l} \Gamma \vdash c : \mathbf{Type} \quad \Gamma \vdash c_{\text{int}} : \kappa \\ \Gamma \vdash c_{\rightarrow} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\ \Gamma \vdash c_{\times} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \end{array}}{\Gamma \vdash \mathbf{Typerec} \ c (c_{\text{int}}, c_{\rightarrow}, c_{\times}) : \kappa}$$

B.2 Constructor Equivalence

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

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

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

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

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

$$\frac{\Gamma \vdash c_1 = c'_1 : \mathbf{Type} \quad \Gamma \vdash c_2 = c'_2 : \mathbf{Type}}{\Gamma \vdash c_1 \rightarrow c_2 = c'_1 \rightarrow c'_2 : \mathbf{Type}}$$

$$\frac{\Gamma \vdash c_1 = c'_1 : \mathbf{Type} \quad \Gamma \vdash c_2 = c'_2 : \mathbf{Type}}{\Gamma \vdash c_1 \times c_2 = c'_1 \times c'_2 : \mathbf{Type}}$$

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

$$\frac{\begin{array}{l} \Gamma \vdash c_{\text{int}} : \kappa \\ \Gamma \vdash c_{\rightarrow} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\ \Gamma \vdash c_{\times} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \end{array}}{\Gamma \vdash \mathbf{Typerec}(\hat{\text{int}}) (c_{\text{int}}, c_{\rightarrow}, c_{\times}) = c_{\text{int}} : \kappa}$$

$$\frac{\begin{array}{l} \Gamma \vdash c_1 : \mathbf{Type} \quad \Gamma \vdash_i c_2 : \mathbf{Type} \quad \Gamma \vdash c_{\text{int}} : \kappa \\ \Gamma \vdash c_{\rightarrow} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\ \Gamma \vdash c_{\times} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \end{array}}{\left\{ \begin{array}{l} \Gamma \vdash \mathbf{Typerec}(c_1 \dot{\rightarrow} c_2) (c_{\text{int}}, c_{\rightarrow}, c_{\times}) = \\ \quad c_{\rightarrow} c_1 c_2 (\mathbf{Typerec} c_1 (c_{\text{int}}, c_{\rightarrow}, c_{\times})) (\mathbf{Typerec} c_2 (c_{\text{int}}, c_{\rightarrow}, c_{\times})) : \kappa \\ \Gamma \vdash \mathbf{Typerec}(c_1 \hat{\times} c_2) (c_{\text{int}}, c_{\rightarrow}, c_{\times}) = \\ \quad c_{\times} c_1 c_2 (\mathbf{Typerec} c_1 (c_{\text{int}}, c_{\rightarrow}, c_{\times})) (\mathbf{Typerec} c_2 (c_{\text{int}}, c_{\rightarrow}, c_{\times})) : \kappa \end{array} \right\}}$$

$$\begin{array}{c}
 \Gamma \vdash c = c' : \mathbf{Type} \\
 \Gamma \vdash c_{\mathbf{int}} = c'_{\mathbf{int}} : \kappa \\
 \Gamma \vdash c_{\rightarrow} = c'_{\rightarrow} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\
 \Gamma \vdash c_{\times} = c'_{\times} : \mathbf{Type} \rightarrow \mathbf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa \\
 \hline
 \Gamma \vdash \mathbf{Typerec} \ c (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) = \mathbf{Typerec} \ c' (c'_{\mathbf{int}}, c'_{\rightarrow}, c'_{\times}) : \kappa
 \end{array}$$

B.3 Type Formation

$\Gamma \vdash \sigma$

$$\begin{array}{c}
 \frac{\Gamma \vdash c : \mathbf{Type}}{\Gamma \vdash T(c)} \\
 \frac{}{\Gamma \vdash \mathbf{int}} \quad \frac{\Gamma \vdash \sigma_1 \quad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \times \sigma_2} \quad \frac{\Gamma \vdash \sigma_1 \quad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2} \\
 \\
 \frac{\Gamma, \alpha : \kappa \vdash \sigma}{\Gamma \vdash \forall \alpha : \kappa. \sigma} \ (\alpha \notin \text{Dom}(\Gamma)) \quad \frac{\Gamma, \alpha : \kappa \vdash \sigma}{\Gamma \vdash \exists \alpha : \kappa. \sigma} \ (\alpha \notin \text{Dom}(\Gamma))
 \end{array}$$

B.3.1 Specific to λ_R

$$\frac{\Gamma \vdash_R c : \mathbf{Type}}{\Gamma \vdash_R R(c)}$$

B.4 Type Equivalence

$\Gamma \vdash \sigma_1 = \sigma_2$

$$\begin{array}{c}
 \frac{\Gamma \vdash c_1 = c_2 : \kappa}{\Gamma \vdash T(c_1) = T(c_2)} \\
 \\
 \frac{}{\Gamma \vdash T(\mathbf{int}) = \mathbf{int}} \quad \frac{}{\Gamma \vdash T(c_1 \hat{\rightarrow} c_2) = T(c_1) \rightarrow T(c_2)} \quad \frac{}{\Gamma \vdash T(c_1 \hat{\times} c_2) = T(c_1) \times T(c_2)} \\
 \\
 \frac{\Gamma \vdash \sigma_1 = \sigma'_1 \quad \Gamma \vdash \sigma_2 = \sigma'_2}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 = \sigma'_1 \rightarrow \sigma'_2} \quad \frac{\Gamma \vdash \sigma_1 = \sigma'_1 \quad \Gamma \vdash \sigma_2 = \sigma'_2}{\Gamma \vdash \sigma_1 \times \sigma_2 = \sigma'_1 \times \sigma'_2} \\
 \\
 \frac{\Gamma, \alpha : \kappa \vdash \sigma = \sigma'}{\Gamma \vdash \forall \alpha : \kappa. \sigma = \forall \alpha : \kappa. \sigma'} \quad \frac{\Gamma, \alpha : \kappa \vdash \sigma = \sigma'}{\Gamma \vdash \exists \alpha : \kappa. \sigma = \exists \alpha : \kappa. \sigma'} \\
 \\
 \frac{}{\Gamma \vdash \sigma = \sigma} \quad \frac{\Gamma \vdash \sigma' = \sigma}{\Gamma \vdash \sigma = \sigma'} \quad \frac{\Gamma \vdash \sigma = \sigma' \quad \Gamma \vdash \sigma' = \sigma''}{\Gamma \vdash \sigma = \sigma''}
 \end{array}$$

B.5 Specific to λ_R

$$\frac{\Gamma \vdash_R c = c' : \mathbf{Type}}{\Gamma \vdash_R R(c) = R(c')}$$

B.6 Term Formation

$\Gamma \vdash e : \sigma$

$$\frac{}{\Gamma \vdash i : \mathbf{int}} \quad \frac{}{\Gamma \vdash x : \sigma} (\Gamma(x) = \sigma)$$

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

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

$$\frac{\Gamma, f:\sigma \vdash e : \sigma \quad \Gamma \vdash \sigma}{\Gamma \vdash \mathbf{fix} f:\sigma. e : \sigma} \left(\begin{array}{l} \sigma = \forall \alpha_1:\kappa_1 \cdots \alpha_n:\kappa_n. \sigma_1 \rightarrow \sigma_2 \\ f \notin \text{Dom}(\Gamma), n \geq 0 \end{array} \right)$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2}$$

$$\frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_1 e : \sigma_1} \quad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_2 e : \sigma_2}$$

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

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

$$\frac{\Gamma, \alpha:\kappa \vdash \sigma_1 : \mathbf{Type} \quad \Gamma \vdash \sigma_2 : \kappa \quad \Gamma \vdash e : \sigma_1[\sigma_2/\alpha]}{\Gamma \vdash \mathbf{pack} e \text{ as } \exists \alpha:\kappa. \sigma_1 \mathbf{hiding} \sigma_2 : \exists \alpha:\kappa. \sigma_1} (\alpha \notin \text{Dom}(\Gamma))$$

$$\frac{\Gamma \vdash e_1 : \exists \alpha:\kappa. \sigma_2 \quad \Gamma, \alpha:\kappa, x:\sigma_2 \vdash e_2 : \sigma_1}{\Gamma \vdash \mathbf{unpack} \langle \alpha, x \rangle = e_1 \text{ in } e_2 : \sigma_1} (\alpha, x \notin \text{Dom}(\Gamma))$$

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

B.6.1 Specific to λ_i^{ML}

$$\frac{\begin{array}{l} \Gamma \vdash_i c : \mathbf{Type} \quad \Gamma \vdash_i e_{\mathbf{int}} : \sigma[\hat{\mathbf{int}}/\alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type} \vdash_i e_{\rightarrow} : \sigma[\beta \hat{\rightarrow} \gamma / \alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type} \vdash_i e_{\times} : \sigma[\beta \hat{\times} \gamma / \alpha] \end{array}}{\Gamma \vdash_i \mathbf{typecase} [\alpha.\sigma] c (e_{\mathbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) : \sigma[c/\alpha]} (\beta, \gamma \notin \text{Dom}(\Gamma))$$

B.6.2 Specific to λ_R

$$\frac{}{\Gamma \vdash_R \mathbf{R}_{\text{int}} : R(\hat{\text{int}})} \quad \frac{\Gamma \vdash_R e_1 : R(c_1) \quad \Gamma \vdash_R e_2 : R(c_2)}{\Gamma \vdash_R \mathbf{R}_{\rightarrow}[c_1, c_2](e_1, e_2) : R(c_1 \hat{\rightarrow} c_2)}$$

$$\frac{\Gamma \vdash_R e_1 : R(c_1) \quad \Gamma \vdash_R e_2 : R(c_2)}{\Gamma \vdash_R \mathbf{R}_{\times}[c_1, c_2](e_1, e_2) : R(c_1 \hat{\times} c_2)}$$

$$\frac{\begin{array}{l} \Gamma, \delta : \mathbf{Type}, \Gamma' \vdash_R e : R(\delta) \\ \Gamma[\hat{\text{int}}/\delta] \vdash_R e_{\text{int}}[\hat{\text{int}}/\delta] : \sigma[\hat{\text{int}}/\delta, \hat{\text{int}}/\alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type}, x : R(\beta), y : R(\gamma), \Gamma'[(\beta \hat{\rightarrow} \gamma)/\delta] \vdash_R \\ \quad e_{\rightarrow}[(\beta \hat{\rightarrow} \gamma)/\delta] : \sigma[(\beta \hat{\rightarrow} \gamma)/\delta, (\beta \hat{\rightarrow} \gamma)/\alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type}, x : R(\beta), y : R(\gamma), \Gamma'[(\beta \hat{\times} \gamma)/\delta] \vdash_R \\ \quad e_{\times}[(\beta \hat{\times} \gamma)/\delta] : \sigma[(\beta \hat{\times} \gamma)/\delta, (\beta \hat{\times} \gamma)/\alpha] \\ \quad (\beta, \gamma, \delta \notin \text{Dom}(\Gamma, \Gamma')) \end{array}}{\Gamma, \delta : \mathbf{Type}, \Gamma' \vdash_R \mathbf{typecase}[\alpha, \sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\delta/\alpha]}$$

$$\frac{\begin{array}{l} \Gamma \vdash_R e : R(c) \quad \Gamma \vdash_R e_{\text{int}} : \sigma[\hat{\text{int}}/\alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type}, x : R(\beta), y : R(\gamma) \vdash_R e_{\rightarrow} : \sigma[\beta \hat{\rightarrow} \gamma/\alpha] \\ \Gamma, \beta : \mathbf{Type}, \gamma : \mathbf{Type}, x : R(\beta), y : R(\gamma) \vdash_R e_{\times} : \sigma[\beta \hat{\times} \gamma/\alpha] \\ \quad (\beta, \gamma \notin \text{Dom}(\Gamma)) \end{array}}{\Gamma \vdash_R \mathbf{typecase}[\delta, \sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[c/\alpha]}$$

$$\frac{\Gamma \vdash_R e : R(\hat{\text{int}}) \quad \Gamma \vdash e_{\text{int}} : \sigma[\hat{\text{int}}/\alpha]}{\Gamma \vdash_R \mathbf{typecase}[\delta, \sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\hat{\text{int}}/\alpha]}$$

$$\frac{\Gamma \vdash_R e : R(c_1 \hat{\rightarrow} c_2) \quad \Gamma, x : R(c_1), y : R(c_2) \vdash_R e_{\rightarrow}[c_1/\beta, c_2/\gamma] : \sigma[(c_1 \hat{\rightarrow} c_2)/\alpha]}{\Gamma \vdash_R \mathbf{typecase}[\delta, \sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[(c_1 \hat{\rightarrow} c_2)/\alpha]}$$

$$\frac{\Gamma \vdash_R e : R(c_1 \hat{\times} c_2) \quad \Gamma, x : R(c_1), y : R(c_2) \vdash_R e_{\times}[c_1/\beta, c_2/\gamma] : \sigma[(c_1 \hat{\times} c_2)/\alpha]}{\Gamma \vdash_R \mathbf{typecase}[\delta, \sigma] e (e_{\text{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[(c_1 \hat{\times} c_2)/\alpha]}$$