*JFP* **12** (6): 567–600, November 2002.  
DOI: 10.1017/S0956796801004282 Printed in the United Kingdom

# *Intensional polymorphism in type-erasure semantics*

KARL CRARY

*Carnegie Mellon University, School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213, USA*

STEPHANIE WEIRICH and GREG MORRISETT

*Department of Computer Science, Cornell University, 4130 Upson Hall, Ithaca, NY 14853-7501, USA*

**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.

**Capsule Review**

Intensional polymorphism has been advocated as a technique for the implementation of polymorphism in functional languages. It is the basis for the TIL compiler and the FLINT backend for SML/NJ. At the heart of intensional polymorphism is passing types to polymorphic functions at run-time, leading to type-passing implementations. Type-passing has a cost, both in terms of complicating the type theory of the compiler intermediate language, as well as the run-time cost of building type representations. This paper mainly addresses the first of these issues, providing an alternative to type-passing, where instead the representations of types are passed at run-time. The payoff for this, for example, is a much simpler typing for closures in a typed compiler intermediate language.

## 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 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 $\alpha$ where $\alpha$ is not only unknown, but may in fact be instantiated by a variety of different types.

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 $\lambda_i^{ML}$, which supports the first-class *intensional analysis*[1] of type information (following earlier work by Constable (Constable, 1982; Constable & Zlatin, 1984)). The $\lambda_i^{ML}$ 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 $\lambda_i^{ML}$ 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 $\lambda_i^{ML}$, it does as one attempts to give $\lambda_i^{ML}$ 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

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

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.

- 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 $\lambda_R$, 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 $\tau$ 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 $\tau$.

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 $\lambda_i^{ML}$), the identity of types cannot be hidden and consequently abstraction is impossible. In contrast, a $\lambda_R$ 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 $\alpha$. In $\lambda_R$, as in most other type systems, $\exists\alpha.\alpha$ implements an abstract type (in this particular example, a useless one), because no representation of $\alpha$ is provided. Dynamic types are implemented in $\lambda_R$ 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 $\lambda_R$ as an extension of Harper and Morrisett's $\lambda_i^{ML}$ 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; $\lambda_i^{ML}$ may be translated in a straightforward syntax-directed manner into $\lambda_R$, as described in section 4.

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

### 1.2 Overview

The remainder of this paper is organized as follows. In section 2 we review the $\lambda_i^{ML}$ calculus. We then present, in Section 3, our $\lambda_R$ 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 $\lambda_i^{ML}$ in $\lambda_R$, 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 32 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 << for shift left, & for bitwise and, and <> 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[α] =
  Typecase α of
    bool => array[int]
    | _ => array[α]

val packedsub : ∀α. PackedArray[α] * int -> α =
  Fn α =>
    typecase α of
      bool => bitsub
      | _ => sub
```

### 2.1 *The $\lambda_i^{ML}$ calculus*

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

The syntax of $\lambda_i^{ML}$, with some minor modifications, appears in figure 1. The

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

Fig. 1. Syntax of $\lambda_i^{ML}$.

$$
(\lambda x{:}\sigma.e)v \mapsto e[v/x] \qquad (\mathbf{fix}\ f{:}\sigma.v)v' \mapsto_i (v[\mathbf{fix}\, f{:}\sigma.v/f])v'
$$

$$
\pi_1\langle v_1, v_2\rangle \mapsto v_1 \qquad \pi_2\langle v_1, v_2\rangle \mapsto v_2
$$

$$
\mathbf{unpack}\ \langle\alpha, x\rangle = (\mathbf{pack}\ v\ \mathbf{as}\ \exists\beta.\sigma\ \mathbf{hiding}\, c)\ \mathbf{in}\ e_2 \mapsto e_2[\sigma_2/\alpha, v/x]
$$

$$
\frac{e_1 \mapsto e_1'}{e_1 e_2 \mapsto e_1' e_2} \qquad \frac{e \mapsto e'}{ve \mapsto ve'} \qquad \frac{e \mapsto e'}{e[c] \mapsto e'[c]}
$$

$$
\frac{e \mapsto e'}{\pi_i e \mapsto \pi_i e'} \qquad \frac{e_1 \mapsto e_1'}{\langle e_1, e_2\rangle \mapsto \langle e_1', e_2\rangle} \qquad \frac{e \mapsto e'}{\langle v, e\rangle \mapsto \langle v, e'\rangle}
$$

$$
\frac{e \mapsto e'}{\mathbf{pack}\ e\ \mathbf{as}\ \exists\beta.\sigma\ \mathbf{hiding}\, c \mapsto \mathbf{pack}\ e'\ \mathbf{as}\ \exists\beta.\sigma\ \mathbf{hiding}\, c}
$$

$$
\frac{e \mapsto e'}{\mathbf{unpack}\ \langle\alpha, x\rangle = e\ \mathbf{in}\ e_2 \mapsto \mathbf{unpack}\ \langle\alpha, x\rangle = e'\ \mathbf{in}\ e_2}
$$

Fig. 2. Operational semantics for core language.

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 $\lambda_i^{ML}$ appears in figures 2 and 3. We write $\mapsto$ for evaluation steps that apply to both $\lambda_i^{ML}$ and $\lambda_R$, and $\mapsto_i$ for evaluation steps that apply only to $\lambda_i^{ML}$. 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 $\lambda_i^{ML}$ is a predicative variant of Girard's $F_\omega$ (1972; 1971) in which the quantified type $\forall\alpha{:}\kappa.\sigma$ ranges only over type constructors and "small" types (*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 **int**, and $T(c_1 \hat{\to} c_2)$ is equal to $T(c_1) \to T(c_2)$.

The type analysis operators are **Typerec** and **typecase** at the constructor and

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

$$(\Lambda\alpha{:}\kappa.e)[c] \mapsto_i e[c/\alpha] \qquad (\textbf{fix}\ f{:}\sigma.v)[c] \mapsto_i (v[\textbf{fix}\, f{:}\sigma.v/f])[c]$$

$$\frac{c \text{ normalizes to } \hat{\textbf{int}}}{\textbf{typecase}\ c\ (e_{\textbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\textbf{int}}}$$

$$\frac{c \text{ normalizes to } (c_1 \hat{\rightarrow} c_2)}{\textbf{typecase}\ c\ (e_{\textbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\rightarrow}[c_1/\beta, c_2/\gamma]}$$

$$\frac{c \text{ normalizes to } (c_1 \hat{\times} c_2)}{\textbf{typecase}\ c\ (e_{\textbf{int}}, \beta\gamma.e_{\rightarrow}, \beta\gamma.e_{\times}) \mapsto_i e_{\times}[c_1/\beta, c_2/\gamma]}$$

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*.

term levels respectively. These operators, given an argument type $c$, dispatch to an appropriate branch based on whether $c$ is $\hat{\textbf{int}}$, 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 $\alpha$ does not appear free in $\sigma$ we often omit it.

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

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

As an example of the use of type analysis in $\lambda_i^{ML}$ (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* $[\hat{\textbf{int}}]\,3$ 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:

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

which returns:

"⟨function, 3⟩"

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

| Judgment | Meaning |
|---|---|
| $\Gamma \vdash c : \kappa$ | $c$ is a valid constructor of kind $\kappa$ |
| $\Gamma \vdash c_1 = c_2 : \kappa$ | $c_1$ and $c_2$ are equal constructors |
| $\Gamma \vdash \sigma$ | $\sigma$ is a valid type |
| $\Gamma \vdash \sigma_1 = \sigma_2$ | $\sigma_1$ and $\sigma_2$ are equal types |
| $\Gamma \vdash e : \sigma$ | $e$ is a term of type $\sigma$ |

Fig. 5. Judgments of $\lambda_i^{ML}$.

When the argument to *tostring* is a product type, the function calls itself recursively. In this branch, the type variables $\beta$ and $\gamma$ 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 $\lambda_i^{ML}$ 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, $\Gamma$ is a unified type and kind context: an ordered, partial map from constructor variables $(\alpha, \beta, \ldots)$ to kinds, and term variables $(x, y, \ldots)$ to types. As before, we use $\vdash$ for rules of these judgments that apply to both $\lambda_i^{ML}$ and $\lambda_R$, employing $\vdash_i$ for rules specific to $\lambda_i^{ML}$.

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

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

Often, to compute the result type $\sigma$ of a **typecase** expression the constructor-level **Typerec** on the argument $\alpha$ 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**

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

Fig. 6. Syntax of $\lambda_R$.

operation on those components.

$$
\begin{array}{l}
\mathbf{Typerec}\,(c_1 \hat{\times} c_2)\,(c_{\mathbf{int}}, c_{\to}, c_{\times}) = \\
\quad c_{\times}\, c_1\, c_2 \\
\qquad (\mathbf{Typerec}\, c_1\, (c_{\mathbf{int}}, c_{\to}, c_{\times})) \\
\qquad (\mathbf{Typerec}\, c_2\, (c_{\mathbf{int}}, c_{\to}, c_{\times}))
\end{array}
$$

The kinding rule for **Typerec** is again the natural one. To compute a constructor of kind $\kappa$, present a type argument and three branches returning $\kappa$ constructors:

$$
\frac{\begin{array}{c}\Gamma \vdash c : \mathbf{Type} \qquad \Gamma \vdash c_{\mathbf{int}} : \kappa \\ \Gamma \vdash c_{\to} : \mathbf{Type} \to \mathbf{Type} \to \kappa \to \kappa \to \kappa \\ \Gamma \vdash c_{\times} : \mathbf{Type} \to \mathbf{Type} \to \kappa \to \kappa \to \kappa\end{array}}{\Gamma \vdash \mathbf{Typerec}\ c\, (c_{\mathbf{int}}, c_{\to}, c_{\times}) : \kappa}
$$

## 3 The $\lambda_R$ calculus

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

### *3.1 Term representations of types*

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

$$(\Lambda\alpha{:}\kappa.v)[c] \mapsto_R (v[c/\alpha])$$

$$(\mathbf{fix}\ f{:}\sigma.v)[c_1]\ldots[c_n]v' \mapsto_R (v[\mathbf{fix}\, f{:}\sigma.v/f])[c_1]\ldots[c_n]v'$$

$$\mathbf{typecase}[\delta.\sigma]\ \mathbf{R}_{\mathbf{int}}\ (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) \mapsto_R e_{\mathbf{int}}$$

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

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

$$\frac{e \mapsto_R e'}{\mathbf{typecase}[\delta.\sigma]\ e\ (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) \mapsto_R \mathbf{typecase}[\delta.\sigma]\ e'\ (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times})}$$

$$\frac{e_1 \mapsto_R e_1'}{\mathbf{R}_{\rightarrow}[c_1,c_2](e_1,e_2) \mapsto_R \mathbf{R}_{\rightarrow}[c_1,c_2](e_1',e_2)} \qquad \frac{e \mapsto_R e'}{\mathbf{R}_{\rightarrow}[c_1,c_2](v,e) \mapsto_R \mathbf{R}_{\rightarrow}[c_1,c_2](v,e')}$$

$$\frac{e_1 \mapsto_R e_1'}{\mathbf{R}_{\times}[c_1,c_2](e_1,e_2) \mapsto_R \mathbf{R}_{\times}[c_1,c_2](e_1',e_2)} \qquad \frac{e \mapsto_R e'}{\mathbf{R}_{\times}[c_1,c_2](v,e) \mapsto_R \mathbf{R}_{\times}[c_1,c_2](v,e')}$$

Fig. 7. Operational semantics for $\lambda_R$.

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 $\beta$ and $\gamma$ 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 **typecase**s 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 $\lambda_R$ is given in figures 2 and 7. Recall that evaluation steps applying to both $\lambda_i^{ML}$ and $\lambda_R$ are written with $\mapsto$. Evaluation steps applying only to $\lambda_R$ 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 $\lambda_i^{ML}$. Aside from the most notable change, the use of type representations in **typecase** expressions, there are other minor changes as well. For example, $\lambda_R$ 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 $\lambda_i^{ML}$, then when viewed under the lens of erasure, $(\mathbf{fix}\, f.v)$ would unroll for no

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

$$\frac{}{\Gamma \vdash_R \mathbf{R_{int}} : R(\hat{\mathbf{int}})} \qquad \frac{\Gamma \vdash_R e_1 : R(\tau_1) \quad \Gamma \vdash_R e_2 : R(\tau_2)}{\Gamma \vdash_R \mathbf{R}_{\rightarrow}[\tau_1, \tau_2](e_1, e_2) : R(\tau_1 \hat{\rightarrow} \tau_2)}$$

$$\frac{\Gamma \vdash_R e_1 : R(\tau_1) \quad \Gamma \vdash e_2 : R(\tau_2)}{\Gamma \vdash_R \mathbf{R}_{\times}[\tau_1, \tau_2](e_1, e_2) : R(\tau_1 \hat{\times} \tau_2)}$$

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

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 $\lambda_R$ to include the $R$ construct, where the representation of a type $\tau$ is given 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 $\tau_1$ and $\tau_2$, 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 $\lambda_R$, the *tostring* function from the previous section can be translated into $\lambda_R$ by requiring it to take an additional term argument, $x_\alpha$ for the representation of the argument type:

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

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 $\lambda_i^{ML}$ often force an inelegant use of **typecase**. In the *tostring* example in section 2, and in its $\lambda_R$ rendition above, we created closures in each of

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

Fig. 9. The variable refining **typecase** rule.

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 as:

$$\begin{array}{l}\mathbf{fix}\ tostring\ :(\forall\alpha{:}\mathbf{Type}\,.\,R(\alpha)\to T(\alpha)\to\mathbf{string}).\\ \quad\Lambda\alpha{:}\mathbf{Type}\,.\,\lambda x_\alpha{:}R(\alpha).\,\lambda obj\,{:}T(\alpha).\\ \quad\quad\mathbf{typecase}[8.\,\mathbf{string}]\ x_\alpha\ \mathbf{of}\\ \quad\quad\quad\mathbf{R_{int}}\Rightarrow \text{int2string}\ obj\\ \quad\quad\quad\mathbf{R_{string}}\Rightarrow obj\\ \quad\quad\quad\mathbf{R}_{\to}[\beta,\gamma](x,y)\Rightarrow\\ \quad\quad\quad\quad\texttt{"function"}\\ \quad\quad\quad\mathbf{R}_{\times}[\beta,\gamma](x,y)\Rightarrow\\ \quad\quad\quad\quad\texttt{"<"}\hat{}\,(tostring\ [\beta]\,x\,(\pi_1\,obj))\hat{}\\ \quad\quad\quad\quad\texttt{","}\hat{}\,(tostring\ [\gamma]\,y\,(\pi_2\,obj))\hat{}\texttt{">"}\end{array}$$

The reason we could not write this function in $\lambda_i^{ML}$ is that it requires the type of *obj* to change based on which branch of the **typecase** is selected. In $\lambda_i^{ML}$, all that is known in the product branch is that *obj* has type $T(\alpha)$; it is not known that it has type $T(\beta\hat{\times}\gamma)$. 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 $\alpha$ 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 $\lambda_R$'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 $\alpha$ to reflect the gain in information. This refinement is done using a simple substitution, as shown in figure 9.

For example, to typecheck the $e_{\to}$ branch, we substitute $\beta\hat{\to}\gamma$ for $\alpha$ everywhere, including the surrounding context.[2] Consequently, the types of the variables bound

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

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

in the context will be refined by that substitution. In contrast, in $\lambda_i^{ML}$ 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.

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

*Theorem 3.1* (*Decidability*)
It is decidable whether or not $\Gamma \vdash e : \tau$ is derivable in $\lambda_R$.

The proof of decidability of $\lambda_R$ typechecking is merely an extension of the decidability of $\lambda_i^{ML}$ 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 & 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 $e \mapsto e'$.

[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.

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

*Lemma 3.4* (*Subject Reduction*)
If $\emptyset \vdash e : \tau$ and $e \mapsto e'$ then $\emptyset \vdash e' : \tau$.

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 $\alpha$, and suppose the refining rule is used to typecheck $e$:

$$\frac{\vdots \qquad \vdots \qquad \cdots}{\Gamma, \alpha{:}\,\mathbf{Type}, \Gamma' \vdash \mathbf{typecase}[\delta.\sigma]\, e'\, (e_{\mathbf{int}}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}) : \sigma[\alpha/\delta]}$$

$$\Gamma, \alpha{:}\,\mathbf{Type}, \Gamma' \vdash e' : R(\alpha) \quad \Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e_{\mathbf{int}}[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha, \hat{\mathbf{int}}/\delta]$$

After substitution for $\alpha$, 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:

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

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

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

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:

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

$$\Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e'[\hat{\mathbf{int}}/\alpha] : R(\hat{\mathbf{int}}) \quad \Gamma, \Gamma'[\hat{\mathbf{int}}/\alpha] \vdash e_{\mathbf{int}}[\hat{\mathbf{int}}/\alpha] : \sigma[\hat{\mathbf{int}}/\alpha][\hat{\mathbf{int}}/\delta]$$

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 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 $e : R(c_1 \hat{\times} c_2)$.

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

*types*

$$
\begin{array}{rcl}
|T(c)| & = & T(c) \\
|\,\mathbf{int}\,| & = & \mathbf{int} \\
|\sigma_1 \to \sigma_2| & = & |\sigma_1| \to |\sigma_2| \\
|\sigma_1 \times \sigma_2| & = & |\sigma_1| \times |\sigma_2| \\
|\forall\alpha{:}\kappa.\sigma| & = & \forall\alpha{:}\kappa.R(\alpha{:}\kappa) \to |\sigma| \\
|\exists\alpha{:}\kappa.\sigma| & = & \exists\alpha{:}\kappa.R(\alpha{:}\kappa) \times |\sigma|
\end{array}
$$

*expressions*

$$
\begin{array}{rcl}
|x| & = & x \\
|i| & = & i \\
|\lambda x{:}\sigma.e| & = & \lambda x{:}|\sigma|.|e| \\
|\,\mathbf{fix}\, f{:}\sigma.v\,| & = & \mathbf{fix}\, f{:}|\sigma|.|v| \\
|e_1 e_2| & = & |e_1||e_2| \\
|\langle e_1, e_2\rangle| & = & \langle |e_1|, |e_2|\rangle \\
|\pi_1 e| & = & \pi_1 |e| \\
|\pi_2 e| & = & \pi_2 |e| \\
|\Lambda\alpha{:}\kappa.e| & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R(\alpha{:}\kappa).|e| \\
|e[c]| & = & |e|\,[c]\,\mathsf{Rep}(c) \\
|\,\mathbf{pack}\ e\ \mathbf{as}\,(\exists\alpha{:}\kappa.\sigma)\,\mathbf{hiding}\,c| & = & \mathbf{pack}\ \langle \mathsf{Rep}(|\sigma_2|), |e|\rangle \\
 & & \quad \mathbf{as}\ \exists\alpha{:}\kappa.\, R(\alpha{:}\kappa) \times |\sigma|\,\mathbf{hiding}\,|c| \\
|\,\mathbf{unpack}\,\langle\alpha, x\rangle = e_1\ \mathbf{in}\ e_2| & = & \mathbf{unpack}\,\langle\alpha, y\rangle = |e_1| \\
 & & \quad \mathbf{in}\ |e_2|[\pi_1 y/x_\alpha, \pi_2 y/x] \\
\left|\begin{array}{l}\mathbf{typecase}[\alpha.\sigma]\ c\ \mathbf{of} \\ \quad \mathbf{int} \Rightarrow e_{\mathbf{int}} \\ \quad \beta \to \gamma \Rightarrow e_{\to} \\ \quad \beta \times \gamma \Rightarrow e_{\times}\end{array}\right| & = & \begin{array}{l}\mathbf{typecase}[\alpha.|\sigma|]\ \mathsf{Rep}(c)\ \mathtt{of} \\ \quad \mathbf{R}_{\mathbf{int}} \Rightarrow |e_{\mathbf{int}}| \\ \quad \mathbf{R}_{\to}[\beta,\gamma](x_\beta, x_\gamma) \Rightarrow |e_{\to}| \\ \quad \mathbf{R}_{\times}[\beta,\gamma](x_\beta, x_\gamma) \Rightarrow |e_{\times}|\end{array}
\end{array}
$$

Fig. 11. Translation from $\lambda_i^{ML}$ to $\lambda_R$.

## 4 Embedding of $\lambda_i^{ML}$

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

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

Creating the representation of a 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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

Fig. 12. Translation of constructors to their representations.

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

Fig. 13. Representations of higher constructors.

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

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

Note that this translation also satisfies the value restriction placed on $\lambda_R$ 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 **Type** $\rightarrow$ **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 $\kappa$, 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 **Typerec** constructor. We represent it as a

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

Fig. 14. Context splitting.

**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:

$$\begin{array}{l} \mathsf{Rep}(\textbf{Typerec}\ \tau(c_{\textbf{int}}, c_\rightarrow, c_\times)) = ((\textbf{fix}\, f\ {:}\ \forall\alpha{:}\textbf{Type}.R(\alpha) \rightarrow R(c^*[\alpha]{:}\kappa). \\ \qquad \Lambda\alpha{:}\textbf{Type}.\lambda x_\alpha{:}R(\alpha). \\ \qquad\quad \textbf{typecase}\ x_\alpha \\ \qquad\qquad R_{\textbf{int}} \Rightarrow \mathsf{Rep}(c_{\textbf{int}}) \\ \qquad\qquad ...) \\ \qquad [\tau]\ \mathsf{Rep}(\tau)) \end{array}$$

where $c^*[\tau] = \textbf{Typerec}\ \tau(c_{\textbf{int}}, c_\rightarrow, c_\times)$ and $\kappa$ 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:

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

The $c_\rightarrow$ arm of the **Typerec** is a function taking four type variables, the first two being $\beta$ and $\gamma$, the second two being the results of calling the **Typerec** recursively on $\beta$ and $\gamma$. 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, $\beta$ and $\gamma$, their representations $x_\beta$ and $x_\gamma$ 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 $\lambda_i^{ML}$ and $\vdash_R$ for typing derivations in $\lambda_R$.

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 $\mathsf{Spl}(\Gamma)$), explicitly adding representations for each variable in the context.

*Lemma 4.1*
If $\Gamma \vdash_R c : \kappa$ then $\mathsf{Spl}(\Gamma) \vdash_R \mathsf{Rep}(c) : R(c : \kappa)$

Now we can establish the static correctness of the embedding:

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\mathsf{Rep}[\![\hat{\mathbf{int}}]\!] & = & \{\mathbf{R}_{\mathbf{int}}\} \\
\mathsf{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 \mathsf{Rep}[\![\tau_i]\!], i = 1, 2\} \\
\mathsf{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 \mathsf{Rep}[\![\tau_i]\!], i = 1, 2\} \\
\mathsf{Rep}[\![\alpha]\!] & = & \{x_\alpha\} \\
\mathsf{Rep}[\![\lambda\alpha{:}\kappa.c]\!] & = & \{\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R(\alpha{:}\kappa).e \mid e \in \mathsf{Rep}[\![c]\!]\} \\
\mathsf{Rep}[\![c_1 c_2]\!] & = & \{e_1[c_2']e_2 \mid e_1 \in \mathsf{Rep}[\![c_1]\!], \Gamma \vdash c_2 = c_2' : \kappa, e_2 \in \mathsf{Rep}[\![c_2]\!]\} \\
\mathsf{Rep}[\![\mathbf{Typerec}\ \tau \quad ]\!] & = & \{(\mathbf{fix}\, f{:}\forall\alpha{:}\mathbf{Type}.R(\alpha) \rightarrow R(c^*[\alpha]{:}\kappa). \\
\quad (c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) & & \quad \Lambda\alpha{:}\mathbf{Type}.\lambda x_\alpha{:}R(\alpha). \\
& & \quad\quad \mathbf{typecase}[R(c^*[\alpha]{:}\kappa)]\ x_\alpha \\
& & \quad\quad\quad \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}} \\
& & \quad\quad\quad \mathbf{R}_{\rightarrow}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
& & \quad\quad\quad\quad e_{\rightarrow}[\beta]x_\beta\ [\gamma]x_\gamma \\
& & \quad\quad\quad\quad\quad [c^*[\beta]](f[\beta]x_\beta)\ [c^*[\gamma]](f[\gamma]x_\gamma) \\
& & \quad\quad\quad \mathbf{R}_{\times}[\beta, \gamma](x_\beta, x_\gamma) \Rightarrow \\
& & \quad\quad\quad\quad e_{\times}[\beta]x_\beta\ [\gamma]x_\gamma \\
& & \quad\quad\quad\quad\quad [c^*[\beta]](f[\beta]x_\beta)\ [c^*[\gamma]](f[\gamma]x_\gamma) \\
& & \quad\quad )\ [\tau']\ e \mid e_{\mathbf{int}} \in \mathsf{Rep}[\![c_{\mathbf{int}}]\!], e_{\rightarrow} \in \mathsf{Rep}[\![c_{\rightarrow}]\!], e_{\times} \in \mathsf{Rep}[\![c_{\times}]\!], \\
& & \quad\quad\quad \Gamma \vdash \tau' = \tau : \mathbf{Type}, e \in \mathsf{Rep}[\![\tau]\!]\ \} \\
& & \quad\quad \text{where } c^*[\tau'] = \mathbf{Typerec}\ \tau'(c_{\mathbf{int}}, c_{\rightarrow}, c_{\times}) \\
& & \quad\quad \text{and } \kappa \text{ is the kind of the full } \mathbf{Typerec} \text{ expression} \\
\\
\overline{\mathsf{Rep}[\![c]\!]} & = & \{e \mid \Gamma \vdash c = c' : \kappa \quad \& \quad e \in \mathsf{Rep}[\![c']\!]\}
\end{array}
$$

Fig. 15. Extended representations

*Theorem 4.2* (*Static correctness*)
Define $|\Gamma|$ as $\mathsf{Spl}(\Gamma')$, where $\Gamma'$ is defined as the pointwise translation of $\Gamma$ (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 $|\Gamma| \vdash_R |e| : |\tau|$

In order to show the dynamic correctness of the embedding, we must show that the result of translation simulates the operation of $\lambda_i^{ML}$. However, because the 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 $\lambda_i^{ML}$ term.

We define the operation $\mathsf{Rep}[\![c]\!]$ which produces a set of representations of the constructor $c$, in Figure 15. For any $c$, $\mathsf{Rep}(c)$ is in the set $\mathsf{Rep}[\![c]\!]$. The other members of this set differ from $\mathsf{Rep}(c)$ only in the embedded constructors. For example, $\mathsf{Rep}[\![\hat{\mathbf{int}} \hat{\rightarrow} \hat{\mathbf{int}}]\!]$ includes both $\mathbf{R}_{\rightarrow}[\hat{\mathbf{int}}, \hat{\mathbf{int}}](\mathbf{R}_{\mathbf{int}}, \mathbf{R}_{\mathbf{int}})$, and $\mathbf{R}_{\rightarrow}[(\lambda\beta{:}\mathbf{Type}.\beta)\hat{\mathbf{int}}, \hat{\mathbf{int}}](\mathbf{R}_{\mathbf{int}}, \mathbf{R}_{\mathbf{int}})$. The set $\overline{\mathsf{Rep}[\![c]\!]}$, defined at the bottom of the figure, is even larger. It includes all representations of equivalent constructors. For

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

*types*

$$
\begin{array}{rcl}
[\![T(c)]\!] & = & \{T(c') \mid \Gamma \vdash c = c' : \mathbf{Type}\} \\
[\![\mathbf{int}]\!] & = & \{\mathbf{int}\} \\
[\![\sigma_1 \to \sigma_2]\!] & = & \{\sigma_1' \to \sigma_2' \mid \sigma_i \in [\![\sigma_i]\!],\ i = 1,2\} \\
[\![\sigma_1 \times \sigma_2]\!] & = & \{\sigma_1' \times \sigma_2' \mid \sigma_i \in [\![\sigma_i]\!],\ i = 1,2\} \\
[\![\forall\alpha{:}\kappa.\sigma]\!] & = & \{\forall\alpha{:}\kappa.R(\alpha{:}\kappa) \to \sigma' \mid \sigma' \in [\![\sigma]\!]\} \\
[\![\exists\alpha{:}\kappa.\sigma]\!] & = & \{\exists\alpha{:}\kappa.R(\alpha{:}\kappa) \times \sigma' \mid \sigma' \in [\![\sigma]\!]\}
\end{array}
$$

*expressions*

$$
\begin{array}{rcl}
[\![x]\!] & = & \{x\} \\
[\![i]\!] & = & \{i\} \\
[\![\lambda x{:}\sigma.e]\!] & = & \{\lambda x{:}\sigma'.e' \mid \sigma' \in [\![\sigma]\!], e' \in [\![e]\!]\} \\
[\![\mathbf{fix}\, f{:}\sigma.v]\!] & = & \{\mathbf{fix}\, f{:}\sigma'.v' \mid \sigma' \in [\![\sigma]\!], v' \in [\![v]\!]\} \\
[\![e_1\ e_2]\!] & = & \{e_1'\ e_2' \mid e_1' \in [\![e_1]\!],\ e_2' \in [\![e_2]\!]\} \\
[\![\langle e_1, e_2\rangle]\!] & = & \{\langle e_1', e_2'\rangle \mid e_1' \in [\![e_1]\!],\ e_2' \in [\![e_2]\!]\} \\
[\![\pi_1 e]\!] & = & \{\pi_1 e' \mid e' \in [\![e]\!]\} \\
[\![\pi_2 e]\!] & = & \{\pi_2 e' \mid e' \in [\![e]\!]\} \\
[\![\Lambda\alpha{:}\kappa.e]\!] & = & \{\Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R(\alpha{:}\kappa).e' \mid e' \in [\![e]\!]\} \\
[\![e[c]]\!] & = & \{e'[c']\ e'' \mid e' \in [\![e]\!], \Gamma \vdash c = c' : \kappa, e'' \in \overline{\mathsf{Rep}[\![c]\!]}\}
\end{array}
$$

$$
\left[\!\!\left[\begin{array}{l}\mathbf{pack}\ e\ \mathbf{as}\ (\exists\alpha{:}\kappa.\sigma) \\ \quad \mathbf{hiding}\ c\end{array}\right]\!\!\right] = \left\{\begin{array}{l|l}\mathbf{pack}\ \langle e'', e'\rangle & e'' \in \overline{\mathsf{Rep}[\![c]\!]} \\ \quad \mathbf{as}\ \exists\alpha{:}\kappa.\, R(\alpha{:}\kappa) \times \sigma' & e' \in [\![e]\!], \sigma' \in [\![\sigma]\!] \\ \quad \mathbf{hiding}\, c' & \Gamma \vdash c' = c : \kappa\end{array}\right\}
$$

$$
[\![\mathbf{unpack}\ \langle\alpha, x\rangle = e_1\ \mathbf{in}\ e_2]\!] = \left\{\begin{array}{l|l}\mathbf{unpack}\ \langle\alpha, y\rangle = e_1' & e_1' \in [\![e_1]\!] \\ \quad \mathbf{in}\ (\lambda x_\alpha{:}R(\alpha{:}\kappa).\lambda x{:}\alpha.e_2')(\pi_1 y)(\pi_2 y) & e_2' \in [\![e_2]\!]\end{array}\right\}
$$

$$
\left[\!\!\left[\begin{array}{l}\mathbf{typecase}[\alpha.\sigma]\ c\ \mathbf{of} \\ \quad \hat{\mathbf{int}} \Rightarrow e_{\mathbf{int}} \\ \quad \beta\hat{\to}\gamma \Rightarrow e_\to \\ \quad \beta \times \gamma \Rightarrow e_\times\end{array}\right]\!\!\right] = \left\{\begin{array}{l|l}\mathbf{typecase}[\alpha.\sigma']\ e\ \mathtt{of} & e\ \in \overline{\mathsf{Rep}[\![c]\!]} \\ \quad \mathbf{R}_{\mathbf{int}} \Rightarrow e_{\mathbf{int}}' & e_{\mathbf{int}}' \in [\![e_{\mathbf{int}}]\!] \\ \quad \mathbf{R}_\to[\beta,\gamma](x_\beta, x_\gamma) \Rightarrow e_\to' & e_\to' \in [\![e_\to]\!] \\ \quad \mathbf{R}_\times[\beta,\gamma](x_\beta, x_\gamma) \Rightarrow e_\times' & e_\times' \in [\![e_\times]\!] \\ & \sigma' \in [\![\sigma]\!]\end{array}\right\}
$$

Fig. 16. Extended translation

example, not only does $\overline{\mathsf{Rep}[\![\hat{\mathbf{int}}\hat{\to}\hat{\mathbf{int}}]\!]}$ include the above terms, but it also includes a representation of $((\lambda\beta{:}\mathbf{Type}.\beta)\,\mathbf{int}) \to \mathbf{int}$

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

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

sets of terms):

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

*Lemma 4.3* (*Substitution*)

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

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 $e \in \overline{\mathsf{Rep}[\![\mathbf{int}]\!]}$ then $e$ must evaluate to $\mathbf{R}_{\mathbf{int}}$ (and similar results for arrow and product types).

*Lemma 4.4*
For all $\emptyset \vdash c : \kappa$, $e \in \mathsf{Rep}[\![c]\!]$ then either $e$ is a value or there exists some $e'$ and $c'$ such that $e \mapsto^+ e'$ and $e' \in \mathsf{Rep}[\![c']\!]$ and $c$ reduces to $c'$.

*Lemma 4.5*
If $e \in \overline{\mathsf{Rep}[\![c]\!]}$ then $e$ evaluates to a value $v \in \overline{\mathsf{Rep}[\![c]\!]}$.

*Corollary 4.6*

1. If $e \in \overline{\mathsf{Rep}[\![\mathbf{int}]\!]}$ then $e$ evaluates to $\mathbf{R}_{\mathbf{int}}$.
2. If $e \in \overline{\mathsf{Rep}[\![\tau_1 \hat{\rightarrow} \tau_2]\!]}$ 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 $v_i \in \overline{\mathsf{Rep}[\![\tau_i]\!]}$ for $i = 1, 2$.
3. If $e \in \overline{\mathsf{Rep}[\![\tau_1 \times \tau_2]\!]}$ 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 $v_i \in \overline{\mathsf{Rep}[\![\tau_i]\!]}$ for $i = 1, 2$.

*Lemma 4.7* (*Simulation*)
If $\vdash_i e_1 : \sigma$ and $e_1 \mapsto_i e_2$ then for all $e_1' \in [\![e_1]\!]$ there exists an $e_2' \in [\![e_2]\!]$ 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, $\lambda_R$-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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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 & 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). So 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 $\tau_1 \to \tau_2$ is translated to a closure of type $\exists\alpha.((\tau_1 \times \alpha) \to \tau_2) \times \alpha$. For example, to closure convert the following function declaration (containing the free variables $x : \mathbf{int}$ and $y : \mathbf{bool}$),

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

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 $y$.

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

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

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

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:

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

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:

$$\begin{aligned}&\lambda\langle f{:}\sigma_f, \langle x{:}\mathbf{int}, y{:}\mathbf{bool}\rangle\rangle.\\&\quad\langle\mathbf{unpack}\,\langle\alpha, f'{:}(\mathbf{int} \times \alpha \to \mathbf{int}) \times \alpha\rangle = f\ \mathbf{in}\,(\pi_1 f')\langle x, \pi_2 f'\rangle, y\rangle\end{aligned}$$

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

existential package:

$$\texttt{val mymonoclosure} = \textbf{pack}\ \langle \texttt{mymonocode}, \langle x, y\rangle\rangle\ \textbf{as}\ \exists\alpha.(\sigma_f \times \alpha \to \textbf{int}) \times \alpha\ \textbf{hiding}\ \textbf{int} \times \textbf{bool}$$

### 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 $\alpha$ and $\beta$ and the free value variables $y : \beta$ and $z : \textbf{int}$, for some appropriately typed term $e_{x,y,z}$):

$$\texttt{val myfunc} : \alpha \to (\textbf{int} \times \beta) = \lambda x{:}\alpha.\, e_{x,y,z}$$

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

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

In pattern matching notation:

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

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

$$\forall\gamma{:}\kappa_{tenv}\,.\,(\pi_1\gamma \times \tau_{venv}) \to (\textbf{int} \times \pi_2\gamma)$$

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

### *5.2.1 Type-passing closures*

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

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

This type uses a translucent type (Harper & Lillibridge, 1994) to dictate that the constructor argument $\gamma$ 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 $\pi_2\gamma = \beta$),

$$\begin{array}{ll} & \forall\gamma{:}\kappa_{tenv}.\,(\pi_1\gamma \times \tau_{venv}) \rightarrow (\textbf{int} \times \pi_2\gamma) \\ \leqslant & \forall\gamma{:}\kappa_{tenv}{=}\langle\alpha,\beta\rangle.\,(\pi_1\gamma \times \tau_{venv}) \rightarrow (\textbf{int} \times \pi_2\gamma) \\ = & \forall\gamma{:}\kappa_{tenv}{=}\langle\alpha,\beta\rangle.\,(\alpha \times \sigma_{venv}) \rightarrow (\textbf{int} \times \beta) \end{array}$$

where $\sigma_{venv} = \beta \times \textbf{int} \cong \tau_{venv}[\langle\alpha,\beta\rangle/\gamma]$. 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:

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

with type:

$$\begin{array}{l} \sigma_{\texttt{myclosure\_2}} = \\ \exists\delta_{tenv}{:}\kappa_{tenv}.\,(\forall\gamma{:}\kappa_{tenv}{=}\delta_{tenv}.\,(\alpha \times \sigma_{venv}) \rightarrow (\textbf{int} \times \beta)) \times \sigma_{venv} \end{array}$$

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

$$\begin{array}{l} \texttt{val myclosure} = \textbf{packkind} \\ \qquad\qquad\qquad\qquad \textbf{pack}\ \texttt{myclosure\_2}\ \textbf{as}\ \ldots\ \textbf{hiding}\ \sigma_{venv} \\ \qquad\qquad\qquad\quad \textbf{as}\ \sigma_{\texttt{myclosure}}\ \textbf{hiding}\ \kappa_{tenv} \end{array}$$

with type:

$$\begin{array}{l} \sigma_{\texttt{myclosure}} = \\ \exists k_{tenv} : \textbf{Kind}.\,\exists\epsilon_{venv} : \textbf{Type}.\,\exists\delta_{tenv}{:}k_{tenv}. \\ \quad (\forall\gamma{:}k_{tenv}{=}\delta_{tenv}.\,(\alpha \times \epsilon_{venv}) \rightarrow (\textbf{int} \times \beta)) \times \epsilon_{venv} \end{array}$$

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

$$\exists k_{tenv} : \mathbf{Kind} \,.\, \exists \epsilon_{venv} : \mathbf{Type} \,.\, \exists \delta_{tenv} : k_{tenv} \,.\, (\forall \gamma : k_{tenv} = \delta_{tenv} \,.\, (\tau_1 \times \epsilon_{venv}) \to \tau_2) \times \epsilon_{venv}$$

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 $\alpha$ and $\beta$ 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

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

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:

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

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:

$$\texttt{val myclosure\_1}' : (\alpha \times \sigma'_{venv}) \to (\mathbf{int} \times \beta) = \texttt{mycode}'[\langle \alpha, \beta \rangle]$$

[4] Except that any function types within $\tau_1$ and $\tau_2$ 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.

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

$$\mathtt{val\ myclosure\_2'} = \langle \mathtt{myclosure\_1'}, \langle x, \langle y, \langle w_\alpha, w_\beta \rangle \rangle \rangle \rangle$$

with type $((\alpha \times \sigma'_{venv}) \to (\mathbf{int} \times \beta)) \times \sigma'_{venv}$.

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

$$\mathtt{val\ myclosure'} = \mathbf{pack}\ \mathtt{myclosure\_2'}\ \mathbf{as}\ \sigma'_{\mathtt{myclosure}}\ \mathbf{hiding}\ \sigma'_{venv}$$

with type:

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

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

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

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 $\alpha$ and $\beta$ 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 $\lambda_R$ type representations are fully first-class. Finally, his system does not support separate compilation well while a transformation to $\lambda_R$ can be applied uniformly across modules.

The issue of type parameter lifting is an important one for compilers based on $\lambda_R$. The construction of type representations at run time would likely lead to significant

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

cost and, in practice, should be lifted out to compile time whenever possible. (Unfortunately, in the presence of polymorphic recursion, which $\lambda_R$ supports, it is not always possible.) Mechanisms for such lifting have been developed by Minamide (in the work discussed above) and by Saha & 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 $\lambda_i^{ML}$. Like $\lambda_i^{ML}$, 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 $\lambda_i^{ML}$ and $\lambda_R$ 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 & 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 transformations 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 $\lambda_i^{ML}$ and $\lambda_R$ 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 $\lambda_i^{ML}$ 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 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 $\lambda_R$. Polymorphism is clearly non-parametric in $\lambda_i^{ML}$, but the lowering of type analysis to explicit term-level representatives makes it plausible that some sort of parametricity could be shown for $\lambda_R$. 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 $\lambda_i^{ML}$; it would be interesting to explore how that intuition may be formalized.

## A Untyped variant of $\lambda_R$

Although the formal static and operational semantics for $\lambda_R$ 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, $\lambda_R{}^\circ$, a translation of $\lambda_R$ 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 $\lambda_R$ 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*

$$
\begin{array}{llll}
(\textit{terms}) & u & ::= & i \mid x \mid \lambda x.u \mid \mathbf{fix}\, f.w \mid u_1 u_2 \\
 & & \mid & \langle u_1, u_2 \rangle \mid \pi_1 u \mid \pi_2 u \mid \mathbf{R}_{\mathbf{int}} \\
 & & \mid & \mathbf{R}_{\rightarrow}(u_1, u_2) \mid \mathbf{R}_{\times}(u_1, u_2) \\
 & & \mid & \mathbf{typecase}\ u\ \mathtt{of} \\
 & & & \quad \mathbf{R}_{\mathbf{int}} \Rightarrow u_{\mathbf{int}} \\
 & & & \quad \mathbf{R}_{\rightarrow}(x, y) \Rightarrow u_{\rightarrow} \\
 & & & \quad \mathbf{R}_{\times}(x, y) \Rightarrow u_{\times} \\
 & & & \\
(\textit{values}) & w & ::= & i \mid \lambda x.u \mid \mathbf{fix}\, f.w \mid \langle w_1, w_2 \rangle \\
 & & \mid & \mathbf{R}_{\mathbf{int}} \mid \mathbf{R}_{\times}(w_1, w_2) \mid \mathbf{R}_{\rightarrow}(w_1, w_2)
\end{array}
$$

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

### A.2 Type erasure

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

### A.3 Operational semantics of $\lambda_R{}^\circ$

$$(\lambda x.u)w \mapsto u[w/x]$$

$$(\mathbf{fix}\ f.w)w' \mapsto (w[\mathbf{fix}\, f.w/f])w'$$

$$\pi_1\langle w_1, w_2\rangle \mapsto w_1 \qquad \pi_2\langle w_1, w_2\rangle \mapsto w_2$$

$$\mathbf{typecase}\ \mathbf{R}_{\mathbf{int}}\ (u_{\mathbf{int}}, xy.u_\rightarrow, xy.u_\times) \mapsto u_{\mathbf{int}}$$

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

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

$$\frac{u_1 \mapsto u_1'}{u_1 u_2 \mapsto u_1' u_2} \qquad \frac{u \mapsto u'}{wu \mapsto wu'}$$

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

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

$$\frac{u_1 \mapsto u_1'}{\mathbf{R}_{\to}(u_1, u_2) \mapsto \mathbf{R}_{\to}(u_1', u_2)} \qquad \frac{u \mapsto u'}{\mathbf{R}_{\to}(w, u) \mapsto \mathbf{R}_{\to}(w, u')}$$

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

$$\frac{u \mapsto u'}{\begin{array}{l}\textbf{typecase}\ u\,(u_{\textbf{int}}, xy.u_{\to}, xy.u_{\times}) \mapsto \\ \quad \textbf{typecase}\ u'\,(u_{\textbf{int}}, xy.u_{\to}, xy.u_{\times})\end{array}}$$

## B Static semantics of $\lambda_i^{ML}$ and $\lambda_R$

### *B.1 Constructor formation*

$\boxed{\Gamma \vdash c : \kappa}$

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

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

$$\frac{\Gamma \vdash c_1 : \textbf{Type} \qquad \Gamma \vdash c_2 : \textbf{Type}}{\Gamma \vdash c_1 \hat{\to} c_2 : \textbf{Type}}$$

$$\frac{\Gamma \vdash c_1 : \textbf{Type} \qquad \Gamma \vdash c_2 : \textbf{Type}}{\Gamma \vdash c_1 \hat{\times} c_2 : \textbf{Type}}$$

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

$$\frac{\Gamma \vdash c_1 : \kappa_1 \to \kappa_2 \qquad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1 c_2 : \kappa_2}$$

$$\frac{\begin{array}{c}\Gamma \vdash c : \textbf{Type} \qquad \Gamma \vdash c_{\textbf{int}} : \kappa \\ \Gamma \vdash c_{\to} : \textbf{Type} \to \textbf{Type} \to \kappa \to \kappa \to \kappa \\ \Gamma \vdash c_{\times} : \textbf{Type} \to \textbf{Type} \to \kappa \to \kappa \to \kappa\end{array}}{\Gamma \vdash \textbf{Typerec}\ c\,(c_{\textbf{int}}, c_{\to}, c_{\times}) : \kappa}$$

### *B.2 Constructor equivalence*

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

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

$$\frac{\Gamma \vdash c : \kappa_1 \to \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c\,\alpha = c : \kappa_1 \to \kappa_2}\ (\alpha \notin Dom(\Gamma))$$

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

$$\frac{\Gamma \vdash c_1 = c_1' : \kappa' \to \kappa \qquad \Gamma \vdash c_2 = c_2' : \kappa'}{\Gamma \vdash c_1 c_2 = c_1' c_2' : \kappa}$$

$$\frac{\Gamma \vdash c_1 = c_1' : \mathbf{Type} \qquad \Gamma \vdash c_2 = c_2' : \mathbf{Type}}{\Gamma \vdash c_1 \to c_2 = c_1' \to c_2' : \mathbf{Type}}$$

$$\frac{\Gamma \vdash c_1 = c_1' : \mathbf{Type} \qquad \Gamma \vdash c_2 = c_2' : \mathbf{Type}}{\Gamma \vdash c_1 \times c_2 = c_1' \times c_2' : \mathbf{Type}}$$

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

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

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

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

### *B.3 Type formation*

$\boxed{\Gamma \vdash \sigma}$

$$\frac{\Gamma \vdash c : \mathbf{Type}}{\Gamma \vdash T(c)}$$

$$\frac{}{\Gamma \vdash \mathbf{int}} \quad \frac{\Gamma \vdash \sigma_1 \qquad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \times \sigma_2} \quad \frac{\Gamma \vdash \sigma_1 \qquad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \to \sigma_2}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \sigma}{\Gamma \vdash \forall \alpha{:}\kappa.\sigma}\ (\alpha \notin Dom(\Gamma)) \quad \frac{\Gamma, \alpha{:}\kappa \vdash \sigma}{\Gamma \vdash \exists \alpha{:}\kappa.\sigma}\ (\alpha \notin Dom(\Gamma))$$

#### *B.3.1 Specific to $\lambda_R$*

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

### *B.4 Type equivalence*

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

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

$$\frac{}{\Gamma \vdash T(\hat{\mathbf{int}}) = \mathbf{int}} \qquad \frac{}{\Gamma \vdash T(c_1 \hat{\rightarrow} c_2) = T(c_1) \rightarrow T(c_2)} \qquad \frac{}{\Gamma \vdash T(c_1 \hat{\times} c_2) = T(c_1) \times T(c_2)}$$

$$\frac{\Gamma \vdash \sigma_1 = \sigma_1' \qquad \Gamma \vdash \sigma_2 = \sigma_2'}{\Gamma \vdash \sigma_1 \rightarrow \sigma_2 = \sigma_1' \rightarrow \sigma_2'} \qquad \frac{\Gamma \vdash \sigma_1 = \sigma_1' \qquad \Gamma \vdash \sigma_2 = \sigma_2'}{\Gamma \vdash \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'}$$

$$\frac{\Gamma, \alpha{:}\kappa \vdash \sigma = \sigma'}{\Gamma \vdash \forall \alpha{:}\kappa.\sigma = \forall \alpha{:}\kappa.\sigma'} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash \sigma = \sigma'}{\Gamma \vdash \exists \alpha{:}\kappa.\sigma = \exists \alpha{:}\kappa.\sigma'}$$

$$\frac{}{\Gamma \vdash \sigma = \sigma} \qquad \frac{\Gamma \vdash \sigma' = \sigma}{\Gamma \vdash \sigma = \sigma'} \qquad \frac{\Gamma \vdash \sigma = \sigma' \qquad \Gamma \vdash \sigma' = \sigma''}{\Gamma \vdash \sigma = \sigma''}$$

### *B.5 Specific to $\lambda_R$*

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

### *B.6 Term formation*

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

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

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

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

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

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

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

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

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

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

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

### *B.6.1 Specific to $\lambda_i^{ML}$*

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

### *B.6.2 Specific to $\lambda_R$*

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

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

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

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

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

https://doi.org/10.1017/S0956796801004282 Published online by Cambridge University Press

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

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

## References


Abadi, M., Banerjee, A., Heintze, N. and Riecke, J. G. (1999) A core calculus of dependency. *26th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 147–160.

Aditya, S. and Caro, A. (1993) Compiler-directed type reconstruction for polymorphic languages. *Conference on Functional Programming Languages and Computer Architecture*, pp. 74–82.

Barendregt, H. P. (1992) Lambda calculi with types. In: Abramsky, S., Gabbay, D. M. and Maibaum, T. S. E. (editors), *Handbook of Logic in Computer Science*, vol. 2. Oxford Science.

Birkedal, L., Tofte, M. and Vejlstrup, M. (1996) From region inference to von Neumann machines via region representation inference. *23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 171–183.

Constable, R. L. (1982) *Intensional analysis of functions and types*. Technical report CSR-118-82, Department of Computer Science, University of Edinburgh.

Constable, R. L. and Zlatin, D. R. (1984) The type theory of PL/CV3. *ACM Trans. Programming Languages & Syst.* **6**(1), 94–117.

Crary, K. and Weirich, S. (1999) Flexible type analysis. *1999 ACM SIGPLAN International Conference on Functional Programming*, pp. 233–248.

Crary, K., Weirich, S. and Morrisett, G. (1998) Intensional polymorphism in type-erasure semantics. *1998 ACM SIGPLAN International Conference on Functional Programming*, pp. 301–312.

Dubois, C., Rouaix, F. and Weis, P. (1995) Extensional polymorphism. *22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 118–129.

Duggan, D. (1998) A type-based semantics for user-defined marshalling in polymorphic languages. *2nd Workshop on Types in Compilation.*

Girard, J.-Y. (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. In: Fenstad, J. E. (editor), *Proceedings 2nd Scandinavian Logic Symposium*, pp. 63–92. North-Holland.

Girard, J.-Y. (1972) *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur*. PhD thesis, Université Paris VII.

Harper, R. and Lillibridge, M. (1993) Explicit polymorphism and CPS conversion. *20th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 206–219.

Harper, R. and Lillibridge, M. (1994) A type-theoretic approach to higher-order modules with sharing. *21st ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 123–137.

Harper, R. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. *22nd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 130–141.

Harper, R., Mitchell, J. C. and Moggi, E. (1990) Higher-order modules and the phase distinction. *17th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 341–354.

Jones, M. P. (1992) A theory of qualified types. *4th European Symposium on Programming: Lecture Notes in Computer Science 582*. Springer-Verlag.

Leroy, X. (1992) Unboxed objects and polymorphic typing. *19th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 177–188.

Minamide, Y. (1997) *Full lifting of type parameters*. Submitted. (Earlier version published as 'Compilation based on a calculus for explicit type-passing', *2nd Fuji International Workshop on Functional and Logic Programming*, 1996.)

Minamide, Y., Morrisett, G. and Harper, R. (1996) Typed closure conversion. *23rd ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 271–283.

Mitchell, J. C. and Plotkin, G. D. (1988) Abstract types have existential type. *ACM Trans. Programming Lang. & Syst.* **10**(3), 470–502.

Morrisett, G., Tarditi, D., Cheng, P., Stone, C., Harper, R. and Lee, P. (1996) (Feb.). The TIL/ML compiler: performance and safety through types. *Workshop on Compiler Support for Systems Software*.

Morrisett, G. (1995) *Compiling with types*. PhD 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, G. and Harper, R. (1997) Semantics of memory management for polymorphic languages. In: Gordon, A. D. and Pitts, A. M. (editors), *Higher Order Operational Techniques in Semantics*. Cambridge University Press.

Morrisett, G., Felleisen, M. and Harper, R. (1995) Abstract models of memory management. *Conference on Functional Programming Languages and Computer Architecture*.

Morrisett, G., Walker, D., Crary, K. and Glew, N. (1999) From System F to typed assembly language. *ACM Trans. Programming Lang. & Syst.* **21**(3), 527–568. (An earlier version appeared in the *1998 Symposium on Principles of Programming Languages*.)

Reynolds, J. C. (1983) Types, abstraction and parametric polymorphism. *Information Processing '83: Proceedings of the IFIP 9th World Computer Congress.*, pp. 513–523. North-Holland.

Ruf, E. (1997) Partitioning dataflow analyses using types. *24th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pp. 15–26.

Saha, B. and Shao, Z. (1998) Optimal type lifting. *2nd Workshop on Types in Compilation*.

Saha, B., Trifonov, V. and Shao, Z. (2000) Fully reflexive intensional type analysis in type erasure semantics. *3rd Workshop on Types in Compilation*.

Shao, Z. (1997a) Flexible representation analysis. *1997 ACM SIGPLAN International Conference on Functional Programming*, pp. 85–98.

Shao, Z. (1997b) 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. and Lee, P. (1996) TIL: a type-directed optimizing compiler for ML. *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 181–192.

Tolmach, A. (1994) Tag-free garbage collection using explicit type parameters. *ACM Conference on Lisp and Functional Programming*, pp. 1–11.

Trifonov, V., Saha, B. and Shao, Z. (2000) Fully reflexive intensional type analysis. *Fifth ACM SIGPLAN International Conference on Functional Programming*, pp. 82–93.

Wright, A. K. and Felleisen, M. (1994) A syntactic approach to type soundness. *Information & Computation*, **115**, 38–94.