Intensional Polymorphism in Type-Erasure Semantics

Karl Crary, Stephanie Weirich and Greg Morrisett

November 13, 1998

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 constructs at the term and type levels, it prevents abstraction, and it severely complicates polymorphic closure conversion.

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

1 Introduction

Type-directed compilers use type information to enable optimizations and transformations that are impossible (or prohibitively difficult) without such information [13, 11, 18, 2, 23, 25, etc.]. However, type-directed compilers for some languages such as Modula-3 and ML face the difficulty that some type information cannot be known at compile time. For example, polymorphic code in ML may operate on inputs of type α where α is not only unknown, but may in fact be instantiated by a variety of different types.

In order to use type information in contexts where it cannot be provided statically, a number of advanced implementation techniques process type information at run time [11, 18, 28, 20, 25]. Such type information is used in two ways: behind the scenes, typically by tag-free garbage collectors [28, 1], and explicitly in program code, for a variety of purposes such as efficient data representation and marshalling [18, 11, 26]. In this paper we focus on the latter area of applications.

To lay a solid foundation for programs that analyze types at run time, Harper and Morrisett devised an internal language, called λ_i^{ML}, that supports the first-class *intensional analysis*[1] of types (following earlier work by Constable [3, 4]). The λ_i^{ML} language and its derivatives were then used extensively in the high-performance ML compilers TIL/ML [27, 17] and FLINT [26]. The primary novelty of λ_i^{ML} is the presence of "typecase" operators at the level of terms and types, that allow computations and type expressions to depend upon the values of other type expressions at run time.

[1]Type analysis is "intensional" when types are analyzed by the structure of their names, rather than by what terms they contain. This approach is critical for practicality.

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

- A type-passing language such as λ_i^{ML} requires that type information *always* be constructed and passed to polymorphic functions. This framework can result in considerable overhead if types are rarely examined at run time, and, as we discuss later, it makes abstraction impossible.
- Type passing results in considerable complexity in language semantics, due in large part to the number of semantic devices that must be duplicated for both terms and types. For example, in semantics that make memory allocation explicit [19, 20] a central device is a formal heap in which data is stored; in a type-erasure framework one such heap suffices, but when types are passed it is necessary to add a second heap (and all the attendant machinery) for type data.
- Type passing also greatly complicates low-level intermediate languages, again due in large part to the duplication of computational devices at the type level, and also to the need to support mixed-phase devices (constructs with both type 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 [21] requires additional instructions for allocating and initializing types, 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 preceding issue, type passing severely complicates typed closure conversion (compare the type-passing system of Minamide *et al.* [15] to the type-erasure system of Morrisett *et al.* [21]). 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 [15].

In this paper we propose a typed calculus, called λ_R, that ameliorates the problems of type passing without sacrificing intensional type analysis. If run-time type dispatch is supported, then clearly on some level types must be passed. The fundamental idea behind our approach is to construct and pass *terms* that represent types instead of the types themselves. The connection between a type τ and its term representation e is made in the static semantics by assigning e the special type $R(\tau)$. Semantically, we may interpret $R(\tau)$ as a singleton type that contains only the representation of τ.

This framework resolves the difficulties with type passing semantics discussed above. In particular, as representations of types are simply terms, we can use the pre-existing term operations to deal with run-time type information in languages and their semantics. Furthermore, we can eliminate the difficulties associated with polymorphic closure conversion, as we show in Section 5. Finally, 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.[2]

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 [22] or through existential types [16]. However, when all types are passed and may be analyzed (as in λ_i^{ML}), the identity of types cannot be hidden and consequently abstraction is impossible. In contrast, a λ_R type can be analyzed only when its representation is available at run time, so abstraction can be achieved simply by not supplying type representations.

[2]In fact, the TIL/ML compiler already finds it necessary to use annotations that mark whether a type must be passed at run-time. Our system provides a formal basis for that mechanism.

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

1.1 Expressiveness

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

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

1.2 Overview

The remainder of this paper is organized as follows: In Section 2 we review the λ_i^{ML} calculus. We then present, in Section 3, our λ_R calculus and discuss its formal semantics, including representation terms, R-types, and the strengthened typecase rule. As examples of its expressiveness, we give an embedding of λ_i^{ML} in λ_R, in Section 4, and in Section 5, we discuss the simplification of polymorphic closure conversion by explicit type passing. We end with 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 operational and static semantics (Appendices B and C.)

2 Intensional Type Analysis

Suppose we wanted to efficiently store an array of boolean values. Most computer architectures require that memory accesses are a word at a time, but it is a terrible waste of space to store booleans as integers. The solution is to pack thirty two booleans into one word and use bit manipulations to retrieve the correct value. To subscript from a packed boolean array, we might use the following function (with `<<` 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 created with intensional type analysis, where in both cases an argument type is examined with a "typecase" form:

```
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 λ_i^{ML} calculus

To formalize the tools of intensional type analysis, we begin by summarizing Harper and Morrisett's λ_i^{ML} calculus [11]. The λ_i^{ML} calculus provides these tools in a form that is relatively simple, but already quite powerful.

The syntax of λ_i^{ML} appears below (modified slightly for presentation). The backbone is a predicative variant of Girard's F_ω [8, 7] 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. The type analysis operators are `Typerec` and `typecase` at the constructor and term levels respectively.

$$
\begin{array}{llll}
(\mathit{kinds}) & \kappa & ::= & \mathsf{Type} \mid \kappa_1 \to \kappa_2 \\[1ex]
(\mathit{con's}) & c & ::= & \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \mathit{int} \mid c_1 \to c_2 \mid c_1 \times c_2 \mid \\
 & & & \mathtt{Typerec}\; c\,(c_{\mathit{int}}, c_{\to}, c_{\times}) \\[1ex]
(\mathit{types}) & \sigma & ::= & c \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \\[1ex]
(\mathit{terms}) & e & ::= & i \mid x \mid \lambda x{:}\sigma.e \mid \mathtt{fix}\, f{:}\sigma.v \mid e_1 e_2 \mid \\
 & & & \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e \mid \Lambda\alpha{:}\kappa.v \mid e[c] \mid \\
 & & & \mathtt{typecase}[\alpha.\sigma]\; c\; \mathtt{of} \\
 & & & \quad \mathit{int} \Rightarrow e_{\mathit{int}} \\
 & & & \quad \beta \to \gamma \Rightarrow e_{\to} \\
 & & & \quad \beta \times \gamma \Rightarrow e_{\times} \\[1ex]
(\mathit{values}) & v & ::= & i \mid \lambda x{:}\sigma.e \mid \mathtt{fix}\, x{:}\sigma.v \mid \langle v_1, v_2\rangle \mid \Lambda\alpha{:}\kappa.v
\end{array}
$$

Figure 1: Syntax of λ_i^{ML}

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

$$\mathtt{typecase}[\alpha.\sigma]\; c\,(e_{\mathit{int}}, \beta\gamma.e_{\to}, \beta\gamma.e_{\times}).$$

As an example of the use of type analysis in λ_i^{ML} (with the addition of another base type, *string*), consider the function *tostring*, presented in Figure 2. This function uses `typecase` to produce a string representation of a data object. For example, the call *tostring* [*int*] 3 returns the string "3". As we cannot provide any information about the implementation of functions, we just return the word "function" when one is encountered, as in the call:

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

```
fix $tostring$ : $(\forall\alpha{:}\mathsf{Type}.\, \alpha \to string)$.
  $\Lambda\alpha{:}\mathsf{Type}$.
    typecase$[\delta.\delta \to string]\, \alpha$ of
      $int \Rightarrow$ int2string
      $string \Rightarrow \lambda obj{:}string.obj$
      $\beta \to \gamma \Rightarrow$
        $\lambda obj{:}(\beta \to \gamma)$."function"
      $\beta \times \gamma \Rightarrow$
        $\lambda obj{:}(\beta \times \gamma)$.
          "<" ^ $(tostring[\beta](\pi_1\, obj))$ ^ "," ^ $(tostring[\gamma](\pi_2\, obj))$ ^ ">"
```

Figure 2: The function *tostring*

which returns

$$\text{“}\langle\text{function, 3}\rangle\text{”}.$$

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

The `typecase` form has a type annotation for type checking without type inference; the annotation $[\alpha.\sigma]$ indicates that given a type constructor argument c, the `typecase` computes a value with type $\sigma[c/\alpha]$ (where this syntax denotes the capture-avoiding substitution of c for α in σ). In this example, each arm returns a function from δ to *string*, where δ is replaced by the appropriate type, such as *int* in the *int* branch, and $\beta \times \gamma$ in the product branch.

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

$$\frac{\begin{array}{l}\Gamma \vdash c : \mathsf{Type} \\ \Gamma, \alpha{:}\mathsf{Type} \vdash \sigma \;\; \text{type} \\ \Gamma \vdash e_{int} : \sigma[int/\alpha] \\ \Gamma, \beta{:}\mathsf{Type}, \gamma{:}\mathsf{Type} \vdash e_{\to} : \sigma[\beta \to \gamma/\alpha] \\ \Gamma, \beta{:}\mathsf{Type}, \gamma{:}\mathsf{Type} \vdash e_{\times} : \sigma[\beta \times \gamma/\alpha]\end{array}}{\Gamma \vdash \left(\begin{array}{l}\mathtt{typecase}[\alpha.\sigma]\; c\; \mathtt{of} \\ \quad int \Rightarrow e_{int} \\ \quad \beta \to \gamma \Rightarrow e_{\to} \\ \quad \beta \times \gamma \Rightarrow e_{\times}\end{array}\right) : \sigma[c/\alpha]}$$

Often, to compute the result type σ of a `typecase` expression the constructor-level `Typerec` on the argument α will be required. `Typerec` allows the creation of new types by similar intensional analysis. Several examples of its use appear in Harper and Morrisett [11], 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 $c_{\to}$ and $c_{\times}$ branches of `Typerec` are constructor functions. For example, if the argument of a `Typerec` operation is $c_1 \times c_2$, then that operation reduces to its $c_{\times}$ branch (a constructor function of four arguments) applied to the components c_1 and c_2, and to the result of recursively computing the `Typerec` operation on those components.

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

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

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

3 The λ_R calculus

Figure 3 presents the syntax of λ_R, which we describe in detail in the following section.

$$\begin{array}{lcll}
(\mathit{kinds}) & \kappa & ::= & \mathsf{Type} \mid \kappa_1 \to \kappa_2 \\[1ex]
(\mathit{con's}) & c & ::= & \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \mathit{int} \mid c_1 \to c_2 \mid c_1 \times c_2 \mid \\
& & & R(c) \mid \texttt{Typerec}\ c\,(c_{int}, c_{\to}, c_{\times}, c_R) \\[1ex]
(\mathit{types}) & \sigma & ::= & c \mid \sigma_1 \to \sigma_2 \mid \sigma_1 \times \sigma_2 \mid \\
& & & \forall\alpha{:}\kappa.\sigma \mid \exists\alpha{:}\kappa.\sigma \\[1ex]
(\mathit{terms}) & e & ::= & i \mid x \mid \lambda x{:}\sigma.e \mid \texttt{fix}\, f{:}\sigma.v \mid e_1 e_2 \mid \langle e_1, e_2\rangle \mid \pi_1 e \mid \pi_2 e \mid \\
& & & \Lambda\alpha{:}\kappa.v \mid e[c] \mid \texttt{pack}\, e\, \texttt{as}\, \exists\alpha.\sigma_1\ \texttt{hiding}\, \sigma_2 \mid \texttt{unpack}\, \langle\alpha, x\rangle = e_1\, \texttt{in}\, e_2 \mid \\
& & & \texttt{R}_{int} \mid \texttt{R}_{\to}(e_1, e_2) \mid \texttt{R}_{\times}(e_1, e_2) \mid \texttt{R}_R(e) \mid \\
& & & \texttt{typecase}[\delta.c]\, e\, \texttt{of} \\
& & & \quad \texttt{R}_{int} \Rightarrow e_{int} \\
& & & \quad \texttt{R}_{\to}(x, y)\, \texttt{as}\, (\beta \to \gamma) \Rightarrow e_{\to} \\
& & & \quad \texttt{R}_{\times}(x, y)\, \texttt{as}\, (\beta \times \gamma) \Rightarrow e_{\times} \\
& & & \quad \texttt{R}_R(x)\, \texttt{as}\, R(\beta) \Rightarrow e_R \\[1ex]
(\mathit{values}) & v & ::= & i \mid \lambda x{:}\sigma.e \mid \texttt{fix}\, f{:}\sigma.v \mid \langle v_1, v_2\rangle \mid \\
& & & \Lambda\alpha{:}\kappa.v \mid v[c] \mid \texttt{pack}\, v\, \texttt{as}\, \exists\alpha.\sigma_1\ \texttt{hiding}\, \sigma_2 \mid \\
& & & \texttt{R}_{int} \mid \texttt{R}_{\to}(v_1, v_2) \mid \texttt{R}_{\times}(v_1, v_2) \mid \texttt{R}_R(v)
\end{array}$$

Figure 3: Syntax of λ_R

3.1 Term Representations of Types

The key feature we add to the term language of λ_R is the representations of types as terms, which remain when the types themselves are ultimately erased. The base type, *int*, has a corresponding representation constant $\texttt{R}_{int}$. Likewise, inductive types have inductively defined representations; the type $\mathit{int} \to \mathit{int}$ is represented by the term $\texttt{R}_{\to}(\texttt{R}_{int}, \texttt{R}_{int})$.

Accordingly, the argument to the term level `typecase` is the representation of a type, instead of a type. For example, if the argument e is of the form $\mathtt{R}_{\rightarrow}(e_1, e_2)$, the arrow branch ($e_{\rightarrow}$) is taken. The type variables β and γ are still bound to the types that e_1 and e_2 represent, but, because we need not only the component types but also their representations, x and y are bound to e_1 and e_2. This notion is reflected in the following rule of the operational semantics:

$$\begin{array}{l}\mathtt{typecase}[\delta.c]\,(\mathtt{R}_{\rightarrow}(e_1,e_2))\,(e_{int}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}, \beta x.e_{R}) \\ \quad\mapsto\ e_{\rightarrow}[\mathcal{D}(e_1), \mathcal{D}(e_2), e_1, e_2/\beta, \gamma, x, y]\end{array}$$

The operation $\mathcal{D}(\cdot)$ in this rule converts a representation to the type that it denotes (Figure 4). The rest of our dynamic semantics is formalized in Appendix B. It is presented as a call-by-value, small step operational semantics.

$$\begin{array}{rcl}\mathcal{D}(\mathtt{R}_{int}) & = & int \\ \mathcal{D}(\mathtt{R}_{\rightarrow}(e_1,e_2)) & = & \mathcal{D}(e_1) \rightarrow \mathcal{D}(e_2) \\ \mathcal{D}(\mathtt{R}_{\times}(e_1,e_2)) & = & \mathcal{D}(e_1) \times \mathcal{D}(e_2) \\ \mathcal{D}(\mathtt{R}_{R}(e)) & = & R(\mathcal{D}(e))\end{array}$$

Figure 4: Translating Representations to Types

In order to assign a type to these representations of types, we have extended the type constructor level of λ_R with the R construct, where the representation of a type τ is given the type $R(\tau)$, and have extended the static semantics accordingly. For example, the formation rule for the representation of function types is

$$\frac{\Gamma \vdash e_1 : R(\tau_1) \quad \Gamma \vdash e_2 : R(\tau_2)}{\Gamma \vdash \mathtt{R}_{\rightarrow}(e_1,e_2) : R(\tau_1 \rightarrow \tau_2)}\ (rep_{\rightarrow})$$

which says that if the two subterms, e_1 and e_2, are type representations of τ_1 and τ_2, then $\mathtt{R}_{\rightarrow}(e_1, e_2)$ will be a representation of $\tau_1 \rightarrow \tau_2$.

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

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

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.

Furthermore, as we have added a new way to form types to the constructor language, we must add another term construct, $\mathtt{R}_R(\cdot)$, to form the representation of representation types. We also extend `typecase` with an extra branch to handle these terms and `Typerec` to handle R-types.

3.2 In-place Refinement of Types

The typing rules of λ_i^{ML} often force an inefficient use of `typecase`. In the *tostring* example in Section 2, we were required to create closures in each of the branches of the `typecase`. It would be more efficient if we could lift the lambdas outside of the `typecase` and have each branch of the `typecase` return a *string*. We could then write this function as:

```
fix $tostring : (\forall\alpha{:}\mathsf{Type}.\ R(\alpha) \to \alpha \to string)$.
  $\Lambda\alpha{:}\mathsf{Type}.\ \lambda x_\alpha{:}R(\alpha).\ \lambda\, obj{:}\alpha.$
    typecase$[\delta.string]\ x_\alpha$ of
      R$_{int} \Rightarrow$ int2string $obj$
      R$_{string} \Rightarrow obj$
      R$_{\to}(x, y)$ as $\beta \to \gamma \Rightarrow$
        "function"
      R$_{\times}(x, y)$ as $\beta \times \gamma \Rightarrow$
        "<"^$(tostring\,[\beta]\,x\,(\pi_1\,obj))$^
        ","^$(tostring\,[\gamma]\,y\,(\pi_2\,obj))$^">"
```

The reason we could not write this function in λ_i^{ML} is that it requires the type of *obj* to change based upon which branch of the `typecase` is selected. In λ_i^{ML}, all that is known in the product branch is that *obj* is of type α, not a tuple. In order to project from it in the recursive calls, the typing rules must update the type of *obj* to reflect the fact that we know that α is $\beta \times \gamma$ in the product branch.

With the right enhancement to the static semantics this optimization is possible. We have held off discussion of the λ_R's `typecase` formation 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 inference rule for type checking a `typecase` term, when the argument is of type $R(\alpha)$, we refine types containing α to reflect the gain in information, as shown below. For simplicity, only some of the rule is given here (the complete rule appears in Appendix C.5):

$$\frac{\begin{array}{l}\Gamma, \alpha{:}\mathsf{Type}, \Gamma' \vdash e : R(\alpha) \\ \Gamma, \Gamma'[int/\alpha] \vdash e_{int}[int/\alpha] : c[int, int/\alpha, \delta] \\ \Gamma, \beta{:}\mathsf{Type}, \gamma{:}\mathsf{Type}, \Gamma'[\beta \to \gamma/\alpha], x{:}R(\beta), y{:}R(\gamma) \vdash e_{\to}[\beta \to \gamma/\alpha] : c[\beta \to \gamma, \beta \to \gamma/\alpha, \delta] \\ \vdots \end{array}}{\Gamma, \alpha{:}\mathsf{Type}, \Gamma' \vdash \mathtt{typecase}[\delta.c]\ e\ (e_{int}, \beta\gamma xy.e_{\to}, \ldots) : c[\alpha/\delta]}$$

Figure 5: The Variable Refining `typecase` Rule (abridged)

For example, to typecheck the $e_{\to}$ branch, we substitute $\beta \to \gamma$ for α everywhere, including the surrounding context.[3] Consequently, the types of the variables bound in the context will be refined by that substitution. Because λ_i^{ML} only makes this substitution in the return type of the branch, and not in the context, in order to propagate this information one must abstract over all variables of interest.

When refinement is not possible we must use a non-refining `typecase` rule reminiscent of λ_i^{ML}, presented in Figure 6.

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

Figure 6: Non-refining `typecase` rule

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

Figure 7: Judgments of λ_R

3.3 Properties of the Formal Semantics

Formally, the static semantics of λ_R consists of a collection of rules for deriving judgments of the forms shown in Figure 7. In these judgments, Γ is a unified type and kind context, mapping constructor variables $(\alpha, \beta, ...)$ to kinds and term variables $(x, y, ...)$ to types. The formal operational and static semantics of λ_R appear in Appendices B and C, and from them we can prove several useful properties about λ_R.

First, we would like to prove the decidability of λ_R typechecking. The only mildly difficult part is equivalence checking for constructors. Based upon the equivalence rules in Appendix C.4 we can define a notion of constructor reduction to a normal form in an obvious manner. This reduction relation can be proved strongly normalizing and confluent (in a manner similar to Morrisett [18]) from which it follows that constructor equivalence is decidable. Therefore we can state the following theorem:

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

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* (where a term that is not a value, and for which no rule of our operational semantics applies, is *stuck*):

Theorem 3.2 (Type Safety) *If $\emptyset \vdash e : \sigma$ and $e \mapsto^* e'$ then e' is not stuck.*

The proof of the Type Safety Theorem is standard, relying on the usual progress, subject reduction and substitution lemmas, listed below.

Lemma 3.3 (Progress) *If $\emptyset \vdash e : \tau$ and e is not a value then there exists an e' such that $e \mapsto e'$.*

Proof of the Progress Lemma is by induction on the derivation of $\emptyset \vdash e : \tau$.

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

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

The proof of the Subject Reduction Lemma is by induction on the derivation of $\emptyset \vdash e : \tau$. As usual, the proof depends on several substitution lemmas; these are shown below. The first five say that constructors may be substituted for type variables in any of our typing judgments. The last says that terms may be substituted for term variables in the term formation judgment.

Lemma 3.5 (Substitution) *1. If $\Gamma, \alpha{:}\kappa' \vdash c : \kappa$ and $\Gamma \vdash c' : \kappa'$ then $\Gamma[c'/\alpha] \vdash c[c'/\alpha] : \kappa$.*

2. If $\Gamma, \alpha{:}\kappa \vdash \sigma$ and $\Gamma \vdash c : \kappa$ then $\Gamma[c/\alpha] \vdash \sigma[c/\alpha]$.

3. If $\Gamma, \alpha{:}\kappa' \vdash c_1 = c_2 : \kappa$ and $\Gamma \vdash c' : \kappa'$ then $\Gamma[c'/\alpha] \vdash c_1[c'/\alpha] = c_2[c'/\alpha] : \kappa$.

4. If $\Gamma, \alpha{:}\kappa \vdash \sigma = \sigma'$ and $\Gamma \vdash c : \kappa$ then $\Gamma[c/\alpha] \vdash \sigma[c/\alpha] = \sigma'[c/\alpha]$.

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

6. If $\Gamma, x{:}\tau' \vdash e : \tau$ and $\emptyset \vdash e' : \tau'$ then $\Gamma \vdash e[e'/x] : \tau$.

The proofs of each of these are fairly straightforward, again by induction on the derivations of the judgments. However, the proof of constructor substitution in terms (Part 5), requires the addition of several trivialization rules to the static semantics of the flavor:

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

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

In each of these rules, the head normal form of the constructor for which the argument to the `typecase` represents is known, so the branch which the `typecase` will step to can be predicted. Therefore only that branch needs to be checked as the others can be considered "dead code".

These rules are needed in the substitution proof in the case where the last rule applied in the derivation was the variable refinement rule (presented in Figure 5), which applies when the argument to a `typecase` term is of type $R(\alpha)$ for some constructor variable α. If we substitute another closed constructor, c of kind Type, for α, we can no longer use the variable refinement rule to type check the term, and, as we have done refinement, the non-refining rule (Figure 6) will not apply either. However, as c is closed and of kind Type, it must be one of int, $\beta \rightarrow \gamma$, $\beta \times \gamma$, or $R(\beta)$ for some β, γ also of kind Type. In each of these cases, we can apply the induction hypothesis to the appropriate branch, and then can apply the appropriate trivialization rule to conclude the desired result.

4 Embedding of λ_i^{ML}

We next present an embedding (written $|\cdot|$) of λ_i^{ML} expressions into λ_R. We do this for two reasons: first, to show that λ_R is as expressive as λ_i^{ML}, and second, to demonstrate a simple use of λ_R as an intermediate language. The main difference between λ_i^{ML} and λ_R is the `typecase` term; in λ_i^{ML} it takes a type constructor as its argument, in λ_R it takes a term representing a type. Therefore, to simulate a λ_i^{ML} `typecase` term with an λ_R `typecase` term, we need to be able to form the term representation of the type constructor argument. We do this in two stages, as shown below. First we translate the type constructor from an λ_i^{ML} constructor to an λ_R constructor, then we translate the λ_R type constructor into its term representation, using an operation written $\Re(\cdot)$.

$$\left|\begin{array}{l}\texttt{typecase}[\delta.\sigma]\, c\, \texttt{of}\\ \quad int \Rightarrow e_{int}\\ \quad \ldots\end{array}\right| = \begin{array}{l}\texttt{typecase}[\delta.\sigma]\, \Re(|c|)\, \texttt{of}\\ \quad \texttt{R}_{int} \Rightarrow |e_{int}|\\ \quad \ldots\end{array}$$

The first part, the translation between λ_i^{ML} constructors and λ_R constructors is fairly simple, as shown in Figure D.1. The only change it makes is to insert a junk result into the branch of the R-constructors, since the source contains no such branch. This junk result is unused, and may therefore be any constructor having the appropriate kind.[4]

$$|\texttt{Typerec}\ c(c_{int}, c_{\to}, c_{\times})| = \texttt{Typerec}\ |c|\,(|c_{int}|\,,|c_{\to}|\,,|c_{\times}|\,, junk)$$

However, creating the representation of a given λ_R type is much more complicated, as can be seen in the definition of $\Re(\cdot)$, shown in Figure D.2. The difficulty lies in that the argument to `Typerec` may contain constructors with free type variables. These type variables are translated to term variables that represent them, but in order to do this, we need to 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" [10] or evidence passing [12]. In the translation of constructor abstractions (at both the constructor and term level), we split the abstractions to take both a constructor and a term variable, where the term variable is constrained to be the representation of that constructor. Application is also changed accordingly:

$$\begin{array}{rcl}\Re(\alpha) &=& x_\alpha\\ \Re(\lambda\alpha{:}\kappa.c) &=& \Lambda\alpha{:}\kappa.\,\lambda x_\alpha{:}R(\alpha{:}\kappa).\,\Re(c)\\ \Re(c_1c_2) &=& \Re(c_1)\,[c_2]\,\Re(c_2)\\ \\ |\Lambda\alpha{:}\kappa.e| &=& \Lambda\alpha{:}\,|\kappa|\,.\lambda x_\alpha{:}R(\alpha{:}\kappa).\,|e|\\ |e[c]| &=& |e|\ [|c|]\,\Re(|c|)\end{array}$$

But, given a type variable, α, what is the type of its corresponding term variable, x_α? If α is of kind Type, then certainly x_α should be of type $R(\alpha)$. But if α is of a higher kind, say, for example, a function from types to types, then x_α should map type representations to type representations, and its type should reflect that fact. For this reason, we introduce the notation $R(c : \kappa)$, the type of the representations of constructor c with kind κ.

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

Figure 8: Representations of higher constructors

If the constructor c is of kind $\kappa_1 \to \kappa_2$, its representation is a polymorphic function that takes the representation of the argument constructor to the representation of the result of applying c to that argument.

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

[4]One such constructor is $\lambda\alpha{:}\mathsf{Type}.\,|c_{\to}|\ \ int\ \alpha\ \ |c_{int}|$.

function:

$$\begin{aligned}\Re(\texttt{Typerec}\ \tau(c_{int}, c_{\to}, c_{\times}, c_R)) = ((&\texttt{fix}\, f : \forall\alpha{:}\mathsf{Type}.R(\alpha) \to R(c^*[\alpha]{:}\kappa).\\ &\Lambda\alpha{:}\mathsf{Type}.\ \lambda x_\alpha{:}R(\alpha).\\ &\quad \texttt{typecase}\ x_\alpha\\ &\qquad R_{int} \Rightarrow \Re(c_{int})\\ &\qquad ...)\\ &[\tau]\,\Re(\tau))\end{aligned}$$

where $c^*[\tau] = \texttt{Typerec}\ \tau(c_{int}, c_{\to}, c_{\times}, c_R)$.

Now what remains are the branches of the `typecase`. In the arrow, product and R branches of the `typecase`, this function must be called recursively on the subcomponents of the type, just as in `Typerec`. For example, consider the arrow case:

$$\begin{aligned}&R_{\to}(x_\beta, x_\gamma)\ \texttt{as}\ (\beta \to \gamma) \Rightarrow\\ &\quad \Re(c_{\to}) \quad [\beta]\ x_\beta \quad [\gamma]\ x_\gamma \quad [c^*[\beta]]\ (f[\beta]\ x_\beta) \quad [c^*[\gamma]]\ (f[\gamma]\ x_\gamma)\end{aligned}$$

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

The static and dynamic correctness of the embedding is not difficult to show, but it requires a formalization of the semantics of λ_i^{ML}, so we omit those theorems here.

5 Polymorphic Typed Closure Conversion

As a final example, we consider typed closure conversion in a λ_R-like framework. The key idea behind closure conversion is to shift from a substitution-based model of execution to an environment-based model via a source-to-source translation. In particular, all functions are replaced with explicit closures which are represented within the language as pairs consisting of a λ-abstraction (the code of the closure), and a tuple (the environment of the closure). The environment contains values for the free variables of the function. The code abstracts the environment as well as the arguments of the function and is thus closed. Hence, the code may be hoisted to the top-level, allocated at compile time, and shared among all substitution instances. Application is rewritten so that the code of a closure is first applied to its environment and then to its arguments.

In the monomorphic case no discrepancy arises between type-passing [15] and type-erasure [21] closure conversion. An existential type is used to hold the type of the closure's environment abstract, so a closure for a $\tau_1 \to \tau_2$ function is given the type $\exists\alpha.((\tau_1 \times \alpha) \to \tau_2) \times \alpha$.

However, with the introduction of polymorphism, significant differences arise between type-passing and type-erasure. The issue stems from the fact that functions may contain free type variables as well as free value variables, and closed code must abstract both. Closures must then provide someway to apply such code to the appropriate type variables. In a type-erasure setting, type application has no run-time effect, so the partial application of code to the appropriate type variables may be performed when closures are created. Consequently, these type variables do not appear in the type of a closure. In fact, closures have the same type ($\exists\alpha.((\tau_1 \times \alpha) \to \tau_2) \times \alpha$) as before.

However, in a type-passing semantics, the application to type arguments is a run-time operation and so such applications must be suspended until the closure is called. Thus, it is necessary for the closure to include a type environment as well as a value environment. The kind of the type environment must be hidden (as did the type of the value environment in the monomorphic case), and the closure's type must enforce the requirement that the code be applied only to the proper type environment (see Minamide *et al.* [15] for detailed explanations of why). The former requires the use of abstract kinds and the latter requires the use of translucent types [9]. This approach results in a closure having the considerably more complicated type (again, see Minamide *et al.* [15] for a formalization of the necessary type theory):

$$\begin{array}{l}\exists k_{\text{tenv}}{:}\mathsf{Kind}.\,\exists \alpha_{\text{venv}}{:}\mathsf{Type}.\,\exists \beta_{\text{tenv}}{:}k_{\text{tenv}}.\\ \quad (\forall \gamma{:}k_{\text{tenv}}{=}\beta_{\text{tenv}}.\,(\tau_1 \times \alpha_{\text{venv}}) \to \tau_2) \times \alpha_{\text{venv}}\end{array}$$

In the above type, k_{tenv} abstracts the kind of the type environment, α_{venv} abstracts the type of the value of the value environment, and β_{tenv} provides the type environment. The code type then takes a type environment γ of kind k_{tenv} as an argument, but γ is constrained (using translucent types) to be the appropriate environment, β_{tenv}.

Since our framework is one of type-erasure, type environments may be resolved by partial application, resulting in the simpler type for closures. However, it is instructive to examine the details. Suppose the function to be closure-converted is the function $f = \lambda x{:}\tau_1.e$ with type $\tau_1 \to \tau_2$ and suppose further that the function contains free occurrences of the type variable α and its representation $x_\alpha{:}R(\alpha)$.

First the function is rewritten in closed form as:

$$\begin{array}{rl} f' & : \forall \alpha.\,(\tau_1 \times R(\alpha)) \to \tau_2 \\ & = \Lambda \alpha.\,\lambda y{:}(\tau_1 \times R(\alpha)).\,e[\pi_1 y, \pi_2 y / x, x_\alpha]\end{array}$$

Then (at run time) f' is instantiated with the type environment (that is, α):

$$f'' : (\tau_1 \times R(\alpha)) \to \tau_2 = f'[\alpha]$$

Finally, a closure is created:

$$\begin{array}{rl} f''' = & \mathtt{pack}\,\langle f'', x_\alpha\rangle\,\mathtt{as}\,\exists \beta.\,((\tau_1 \times \beta) \to \tau_2) \times \beta \\ & \mathtt{hiding}\,R(\alpha)\end{array}$$

Consider what has become of the mechanisms for type-passing closure conversion: The type of f'' requires that it be applied (for its second argument) only to the representation of α. So the translucency mechanism appears again, suggesting that translucency is inherent in type-passing closure conversion. However, this version of translucency has two advantages; the necessary type theory is simpler, and the translucency is completely hidden by the existential packaging in the eventual closure. On the other hand, abstract kinds do not appear in the process, suggesting them to be an artifact of true type-passing.

6 Related Work

Closely related to our work is the work of Minamide on lifting of type parameters for tag-free garbage collection [14]. 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 [12]) and ordinary terms, while λ_R type representations are fully first-class.

The issue of type parameter lifting is an important one for compilers based on λ_R. The construction of type representations at run time would likely lead to significant cost and, in practice, should be lifted out to compile time whenever possible. (Unfortunately, in the presence of polymorphic recursion, which λ_R supports, it is not always possible.) Mechanisms for such lifting have been developed by Minamide (in the work discussed above) and by Saha and Shao [24].

Dubois *et al.* [5] 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 [6] proposes another typed framework for intensional type analysis that is similar in some ways to λ_i^{ML}. Like λ_i^{ML}, Duggan's system passes types implicitly and allows for the intensional analysis of types at the term level. Duggan's system does not support intensional type analysis at the constructor level, as λ_i^{ML} and λ_R do, but it adds a facility for defining type classes (using union and recursive kinds) and allows type analysis to be restricted to members of such classes.

7 Conclusions and Future Directions

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

For example, Morrisett *et al.* [21] 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 [27, 17] and FLINT [26], and has made it infeasible to use intensional type analysis in the end-to-end typed compiler TALC [21]. The framework in this paper makes it possible to unify these two lines of work for the first time.

In pursuance of this aim, an important direction for future work is to extend the mechanisms of λ_R into lower-level typed intermediate languages such as typed assembly language [21]. Among the issues to be explored in such research is how to analyze the more complicated types used in typed assembly language, and how to perform type-directed dispatch without an atomic `typecase` construct. Another issue to explore is analysis of quantified types and whether such mechanisms are useful in practice.

Another important question is whether a parametricity theorem like that of Reynolds [22] can be shown for λ_R. Polymorphism is clearly non-parametric in λ_i^{ML}, but the lowering of type analysis to explicit term-level representatives makes it plausible that some sort of parametricity could be shown for λ_R. In other words, we discussed at an intuitive level in Section 1 how the explicit passing of types restores the ability to abstract types that was discarded by λ_i^{ML}; it would be interesting to explore how that intuition may be formalized.

References

[1] Shail Aditya and Alejandro Caro. Compiler-directed type reconstruction for polymorphic languages. In *Conference on Functional Programming Languages and Computer Architecture*, pages 74–82, Copenhagen, June 1993.

[2] Lars Birkedal, Mads Tofte, and Magnus Vejlstrup. From region inference to von Neumann machines via region representation inference. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 171–183, St. Petersburg, January 1996.

[3] Robert L. Constable. Intensional analysis of functions and types. Technical Report CSR-118-82, Department of Computer Science, University of Edinburgh, June 1982.

[4] Robert L. Constable and Daniel R. Zlatin. The type theory of PL/CV3. *ACM Transactions on Programming Languages and Systems*, 6(1):94–117, January 1984.

[5] Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Twenty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, January 1995.

[6] Dominic Duggan. A type-based semantics for user-defined marshalling in polymorphic languages. In *Second Workshop on Types in Compilation*, March 1998.

[7] Jean-Yves Girard. 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 J. E. Fenstad, editor, *Proceedings of the Second Scandinavian Logic Symposium*, pages 63–92. North-Holland Publishing Co., 1971.

[8] Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972.

[9] Robert Harper and Mark Lillibridge. A type-theoretic approach to higher-order modules with sharing. In *Twenty-First ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 123–137, Portland, Oregon, January 1994.

[10] Robert Harper, John C. Mitchell, and Eugenio Moggi. Higher-order modules and the phase distinction. In *Seventeenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 341–354, San Francisco, January 1990.

[11] Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995.

[12] Mark P. Jones. A theory of qualified types. In *Fourth European Symposium on Programming*, volume 582 of *Lecture Notes in Computer Science*, Rennes, France, 1992. Springer-Verlag.

[13] Xavier Leroy. Unboxed objects and polymorphic typing. In *Nineteenth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 177–188, 1992.

[14] Yasuhiko Minamide. Full lifting of type parameters. Submitted for publication. Earlier version published as "Compilation Based on a Calculus for Explicit Type-Passing" in the *Second Fuji International Workshop on Functional and Logic Programming,* 1996., 1997.

[15] Yasuhiko Minamide, Greg Morrisett, and Robert Harper. Typed closure conversion. In *Twenty-Third ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 271–283, St. Petersburg, Florida, January 1996.

[16] John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988.

[17] G. Morrisett, D. Tarditi, P. Cheng, C. Stone, R. Harper, and P. Lee. The TIL/ML compiler: Performance and safety through types. In *Workshop on Compiler Support for Systems Software*, Tucson, February 1996.

[18] Greg Morrisett. *Compiling with Types.* PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995.

[19] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *Conference on Functional Programming Languages and Computer Architecture*, 1995.

[20] Greg Morrisett and Robert Harper. Semantics of memory management for polymorphic languages. In A. D. Gordon and A. M. Pitts, editors, *Higher Order Operational Techniques in Semantics*. Cambridge University Press, 1997.

[21] Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From System F to typed assembly language. In *Twenty-Fifth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 85–97, San Diego, January 1998. Extended version published as Cornell University technical report TR97-1651.

[22] John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress.

[23] Erik Ruf. Partitioning dataflow analyses using types. In *Twenty-Fourth ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, pages 15–26, Paris, January 1997.

[24] Bratin Saha and Zhong Shao. Optimal type lifting. In *Second Workshop on Types in Compilation*, March 1998.

[25] Zhong Shao. Flexible representation analysis. In *1997 ACM SIGPLAN International Conference on Functional Programming*, pages 85–98, Amsterdam, June 1997.

[26] Zhong Shao. An overview of the FLINT/ML compiler. In *1997 Workshop on Types in Compilation*, Amsterdam, June 1997. ACM SIGPLAN. Published as Boston College Computer Science Department Technical Report BCCS-97-03.

[27] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. In *1996 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 181–192, May 1996.

[28] Andrew Tolmach. Tag-free garbage collection using explicit type parameters. In *ACM Conference on Lisp and Functional Programming*, pages 1–11, Orlando, June 1994.

A Untyped Calculus

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

Theorem A.1 *1. If $e_1 \mapsto^* e_2$ then ${e_1}^\circ \mapsto^* {e_2}^\circ$.*

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

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

Corollary A.2 *If $\emptyset \vdash e : \tau$ and $e^\circ \mapsto^* u$ then u is not stuck.*

A.1 Syntax of Untyped Calculus

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

$$\begin{array}{llll} (\mathit{values}) & w & ::= & i \mid \lambda x.u \mid \mathtt{fix}\, f.w \mid \langle w_1, w_2\rangle \mid \\ & & & \mathtt{R}_{int} \mid \mathtt{R}_{\times}(w_1,w_2) \mid \mathtt{R}_{\rightarrow}(w_1,w_2) \mid \\ & & & \mathtt{R}_{R}(w) \end{array}$$

A.2 Type Erasure

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

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

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

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

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

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

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

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

$$\texttt{typecase}\,(\texttt{R}_R(w))\,(u_{int}, xy.u_\to, xy.u_\times, x.u_R) \mapsto u_R[w/x]$$

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

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

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

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

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

$$\frac{u \mapsto u'}{\texttt{R}_R(u) \mapsto \texttt{R}_R(u')}$$

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

B Operational Semantics

$$(\lambda x{:}c.e)v \mapsto e[v/x]$$

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

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

$$(\texttt{fix}\ f{:}c.v)v' \mapsto (v[\texttt{fix}\,f{:}c.v/f])v'$$

$$(\texttt{fix}\ f{:}c.v)[c'] \mapsto (v[\texttt{fix}\,f{:}c.v/f])[c']$$

$$\begin{array}{l}\texttt{unpack}\,\langle\alpha, x\rangle = (\texttt{pack}\,v\,\texttt{as}\,\exists\beta.c_1\ \texttt{hiding}\,c_2) \\ \quad \texttt{in}\,e_2 \mapsto e_2[c_2, v/\alpha, x]\end{array}$$

$$\begin{array}{l}\texttt{typecase}[\delta.c]\,\texttt{R}_{int}\,(e_{int}, \beta\gamma xy.e_\to, \\ \quad \beta\gamma xy.e_\times, \beta x.e_R) \mapsto e_{int}\end{array}$$

$$\begin{array}{l}\texttt{typecase}[\delta.c]\,(\texttt{R}_\to(v_1, v_2))\,(e_{int}, \beta\gamma xy.e_\to, \\ \quad \beta\gamma xy.e_\times, \beta x.e_R) \mapsto e_\to[\mathcal{D}(v_1), \mathcal{D}(v_2), v_1, v_2/\beta, \gamma, x, y]\end{array}$$

$$\begin{array}{l}\texttt{typecase}[\delta.c]\,(\texttt{R}_\times(v_1, v_2))\,(e_{int}, \beta\gamma xy.e_\to, \\ \quad \beta\gamma xy.e_\times, \beta x.e_R) \mapsto e_\times[\mathcal{D}(v_2), \mathcal{D}(v_2), v_1, v_2/\beta, \gamma, x, y]\end{array}$$

$$\begin{array}{l}\texttt{typecase}[\delta.c]\,(\texttt{R}_R(v))\,(e_{int}, \beta\gamma xy.e_{\rightarrow},\\ \quad \beta\gamma xy.e_{\times}, \beta x.e_R) \mapsto e_R[\mathcal{D}(v), v/\beta, x]\end{array}$$

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

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

$$\frac{e \mapsto e'}{\texttt{pack}\, e\, \texttt{as}\, \exists\beta.c_1\ \texttt{hiding}\, c_2 \mapsto \texttt{pack}\, e'\, \texttt{as}\, \exists\beta.c_1\ \texttt{hiding}\, c_2}$$

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

$$\frac{e \mapsto e'}{\begin{array}{l}\texttt{typecase}[\delta.\sigma]\, e\, (e_{int}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}, \beta x.e_R) \mapsto\\ \quad \texttt{typecase}[\delta.\sigma]\, e'\, (e_{int}, \beta\gamma xy.e_{\rightarrow}, \beta\gamma xy.e_{\times}, \beta x.e_R)\end{array}}$$

$$\frac{e_1 \mapsto e_1'}{\texttt{R}_{\rightarrow}(e_1, e_2) \mapsto \texttt{R}_{\rightarrow}(e_1', e_2)}$$

$$\frac{e \mapsto e'}{\texttt{R}_{\rightarrow}(v, e) \mapsto \texttt{R}_{\rightarrow}(v, e')} \qquad \frac{e_1 \mapsto e_1'}{\texttt{R}_{\times}(e_1, e_2) \mapsto \texttt{R}_{\times}(e_1', e_2)}$$

$$\frac{e \mapsto e'}{\texttt{R}_{\times}(v, e) \mapsto \texttt{R}_{\times}(v, e')} \qquad \frac{e \mapsto e'}{\texttt{R}_R(e) \mapsto \texttt{R}_R(e')}$$

C Static Semantics

C.1 Constructor Formation

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

$$\frac{}{\Gamma \vdash \mathit{int} : \mathsf{Type}} \qquad \frac{}{\Gamma \vdash \alpha : \kappa}\ (\Gamma(\alpha) = \kappa) \qquad \frac{\Gamma \vdash c_1 : \mathsf{Type} \qquad \Gamma \vdash c_2 : \mathsf{Type}}{\Gamma \vdash c_1 \rightarrow c_2 : \mathsf{Type}}$$

$$\frac{\Gamma \vdash c_1 : \mathsf{Type} \qquad \Gamma \vdash c_2 : \mathsf{Type}}{\Gamma \vdash c_1 \times c_2 : \mathsf{Type}} \qquad \frac{\Gamma, \alpha{:}\kappa_1 \vdash c : \kappa_2}{\Gamma \vdash \lambda\alpha{:}\kappa_1.c : \kappa_1 \rightarrow \kappa_2}$$

$$\frac{\Gamma \vdash c_1 : \kappa_1 \rightarrow \kappa_2 \qquad \Gamma \vdash c_2 : \kappa_1}{\Gamma \vdash c_1 c_2 : \kappa_2}\ (\alpha \notin \mathit{Dom}(\Gamma)) \qquad \frac{\Gamma \vdash c : \mathsf{Type}}{\Gamma \vdash R(c) : \mathsf{Type}}$$

$$\frac{\begin{array}{c}\Gamma \vdash c : \mathsf{Type} \qquad \Gamma \vdash c_{int} : \kappa\\ \Gamma \vdash c_{\rightarrow} : \mathsf{Type} \rightarrow \mathsf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa\\ \Gamma \vdash c_{\times} : \mathsf{Type} \rightarrow \mathsf{Type} \rightarrow \kappa \rightarrow \kappa \rightarrow \kappa\\ \Gamma \vdash c_R : \mathsf{Type} \rightarrow \kappa \rightarrow \kappa\end{array}}{\Gamma \vdash \texttt{Typerec}\ c(c_{int}, c_{\rightarrow}, c_{\times}, c_R) : \kappa}$$

C.2 Type Formation

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

$$\frac{\Gamma \vdash c : \mathsf{Type}}{\Gamma \vdash c} \qquad \frac{\Gamma \vdash \sigma_1 \qquad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash \sigma_1 \qquad \Gamma \vdash \sigma_2}{\Gamma \vdash \sigma_1 \to \sigma_2}$$

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

C.3 Constructor Equivalence

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

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

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

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

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

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

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

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

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

C.4 Type Equivalence

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

$$\frac{\Gamma \vdash \sigma_1 = \sigma_2 : \kappa}{\Gamma \vdash \sigma_1 = \sigma_2} \qquad \frac{\Gamma \vdash \sigma_1 = \sigma_1' \qquad \Gamma \vdash \sigma_2 = \sigma_2'}{\Gamma \vdash \sigma_1 \to \sigma_2 = \sigma_1' \to \sigma_2'} \qquad \frac{\Gamma \vdash \sigma_1 = \sigma_1' \qquad \Gamma \vdash \sigma_2 = \sigma_2'}{\Gamma \vdash \sigma_1 \times \sigma_2 = \sigma_1' \times \sigma_2'}$$

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

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

C.5 Term Formation

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

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

$$\frac{\Gamma, x{:}\sigma_2 \vdash e : \sigma_1 \quad \Gamma \vdash \sigma_2}{\Gamma \vdash \lambda x{:}\sigma_2.e : \sigma_2 \to \sigma_1}\ (x \notin \mathit{Dom}(\Gamma)) \qquad \frac{\Gamma \vdash e_1 : \sigma_2 \to \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash e_1 e_2 : \sigma_1}$$

$$\frac{\Gamma, f{:}\sigma \vdash e : \sigma \quad \Gamma \vdash \sigma}{\Gamma \vdash \mathtt{fix}\ f{:}\sigma.\, e : \sigma}\ (f \notin \mathit{Dom}(\Gamma), \sigma = \forall \alpha_1{:}\kappa_1 \cdots \alpha_n{:}\kappa_n.\sigma_1 \to \sigma_2,\ n \geq 0)$$

$$\frac{\Gamma \vdash e_1 : \sigma_1 \quad \Gamma \vdash e_2 : \sigma_2}{\Gamma \vdash \langle e_1, e_2 \rangle : \sigma_1 \times \sigma_2} \qquad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_1 e : \sigma_1} \qquad \frac{\Gamma \vdash e : \sigma_1 \times \sigma_2}{\Gamma \vdash \pi_2 e : \sigma_2}$$

$$\frac{\Gamma \vdash e : \forall \alpha{:}\kappa.\sigma \quad \Gamma \vdash c : \kappa}{\Gamma \vdash e[c] : \sigma[c/\alpha]} \qquad \frac{\Gamma, \alpha{:}\kappa \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha{:}\kappa.e : \forall \alpha{:}\kappa.\sigma}\ (x \notin \mathit{Dom}(\Gamma))$$

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

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

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

$$\frac{}{\Gamma \vdash R_{int} : R(int)} \qquad \frac{\Gamma \vdash e_1 : R(c_1) \quad \Gamma \vdash e_2 : R(c_2)}{\Gamma \vdash \mathtt{R}_{\to}(e_1, e_2) : R(c_1 \to c_2)}$$

$$\frac{\Gamma \vdash e_1 : R(c_1) \quad \Gamma \vdash e_2 : R(c_2)}{\Gamma \vdash \mathtt{R}_{\times}(e_1, e_2) : R(c_1 \times c_2)} \qquad \frac{\Gamma \vdash e : R(c)}{\Gamma \vdash \mathtt{R}_{R}(e) : R(R(c))}$$

$$\frac{\begin{array}{c} \Gamma, \alpha{:}\mathsf{Type}, \Gamma' \vdash e : R(\alpha) \\ \Gamma(\Gamma'[int/\alpha]) \vdash e_{int}[int/\alpha] : \sigma[int, int/\alpha, \delta] \\ \Gamma, \beta{:}\mathsf{Type}, \gamma{:}\mathsf{Type}, (\Gamma'[\beta \to \gamma/\alpha]), x{:}R(\beta), y{:}R(\gamma) \\ \vdash e_{\to}[\beta \to \gamma/\alpha] : \sigma[\beta \to \gamma, \beta \to \gamma/\alpha, \delta] \\ \Gamma, \beta{:}\mathsf{Type}, \gamma{:}\mathsf{Type}, (\Gamma'[\beta \times \gamma/\alpha]), x{:}R(\beta), y{:}R(\gamma) \\ \vdash e_{\times}[\beta \times \gamma/\alpha] : \sigma[\beta \times \gamma, \beta \times \gamma/\alpha, \delta] \\ \Gamma, \beta{:}\mathsf{Type}, (\Gamma'[R(\beta)/\alpha]), x{:}R(\beta) \vdash e_{R}[R(\beta)/\alpha] : \sigma[R(\beta), R(\beta)/\alpha, \delta] \\ \hfill (\alpha, \beta, \gamma \notin Dom(\Gamma, \Gamma')) \end{array}}{\begin{array}{l} \Gamma, \alpha{:}\mathsf{Type}, \Gamma' \\ \quad \vdash \mathtt{typecase}[\delta.\sigma]\, e\, (e_{int}, \beta\gamma xy.e_{\to}, \beta\gamma xy.e_{\times}, \beta x.e_{R}) : \sigma[\alpha/\delta] \end{array}}$$

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

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

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

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

$$\frac{\Gamma \vdash e : R(R(c)) \quad \Gamma, x{:}R(c) \vdash e_{R}[c/\beta] : \sigma[R(c)/\delta]}{\Gamma \vdash \mathtt{typecase}[\delta.\sigma]\, e\, (e_{int}, \beta\gamma xy.e_{\to}, \beta\gamma xy.e_{\times}, \beta x.e_{R}) : \sigma[R(c)/\delta]}$$

D Translation of λ_i^{ML} to λ_R

D.1 Translation of λ_i^{ML} expressions to λ_R

$$
\begin{array}{lrcl}
\textit{kinds} & |\kappa| & = & \kappa \\
\\
\textit{constructors} & |\alpha| & = & \alpha \\
 & |\lambda\alpha{:}\kappa.c| & = & \lambda\alpha{:}|\kappa|.|c| \\
 & |c_1 c_2| & = & |c_1||c_2| \\
 & |\,\mathtt{Typerec}\ c(c_{int}, c_{\to}, c_{\times})| & = & \mathtt{Typerec}\ |c|(|c_{int}|, |c_{\to}|, |c_{\times}|, \\
 & & & \quad \lambda\alpha{:}\mathsf{Type}.\ |c_{\to}|\ \ \mathit{int}\ \alpha\ \ |c_{int}|) \\
 & |\mathit{int}| & = & \mathit{int} \\
 & |c_1 \to c_2| & = & |c_1| \to |c_2| \\
 & |c_1 \times c_2| & = & |c_1| \times |c_2| \\
\\
\textit{types} & |c| & = & |c| \\
 & |\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| \\
\\
\textit{expressions} & |x| & = & x \\
 & |i| & = & i \\
 & |\lambda x{:}\sigma.e| & = & \lambda x{:}\,|\sigma|\,.\,|e| \\
 & |\Lambda\alpha{:}\kappa.e| & = & \Lambda\alpha{:}\,|\kappa|\,.\lambda x_{\alpha}{:}R(\alpha{:}\kappa).\,|e| \\
 & |e_1 e_2| & = & |e_1|\,|e_2| \\
 & |e[c]| & = & |e|\ [|c|]\,\Re(|c|) \\
 & |\mathtt{fix}\ f{:}\sigma.v| & = & \mathtt{fix}\ f{:}\,|\sigma|\,.\,|v| \\
 & |\langle e_1, e_2\rangle| & = & \langle |e_1|\,, |e_2|\rangle \\
 & |\pi_1 e| & = & \pi_1\,|e| \\
 & |\pi_2 e| & = & \pi_2\,|e| \\
 & \left|\begin{array}{l} \mathtt{typecase}[\delta.\sigma]\ c\ \mathtt{of} \\ \quad \mathit{int} \Rightarrow e_{int} \\ \quad \beta \to \gamma \Rightarrow e_{\to} \\ \quad \beta \times \gamma \Rightarrow e_{\times} \end{array}\right| & = & \begin{array}{l} \mathtt{typecase}[\delta.\sigma]\ \Re(|c|)\ \mathtt{of} \\ \quad \mathtt{R}_{int} \Rightarrow |e_{int}| \\ \quad \mathtt{R}_{\to}(x_{\beta}, x_{\gamma})\ \mathsf{as}\ (\beta \to \gamma) \Rightarrow |e_{\to}| \\ \quad \mathtt{R}_{\times}(x_{\beta}, x_{\gamma})\ \mathsf{as}\ (\beta \times \gamma) \Rightarrow |e_{\times}| \\ \quad \mathtt{R}_{R}(x_{\beta})\ \mathsf{as}\ R(\beta) \Rightarrow |e_{int}| \end{array}
\end{array}
$$

D.2 Translation of λ_R constructors to their representations.

$$
\begin{array}{rcl}
\Re(int) & = & R_{int} \\
\Re(\tau_1 \to \tau_2) & = & R_{\to}(\Re(\tau_1), \Re(\tau_2)) \\
\Re(\tau_1 \times \tau_2) & = & R_{\times}(\Re(\tau_1), \Re(\tau_2)) \\
\Re(R(\tau)) & = & R_{R}(\Re(\tau)) \\
\Re(\alpha) & = & x_\alpha \\
\Re(\lambda\alpha{:}\kappa.c) & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}R(\alpha{:}\kappa).\Re(c) \\
\Re(c_1 c_2) & = & \Re(c_1)[c_2]\Re(c_2) \\
\Re(\texttt{Typerec}\ \tau(c_{int}, c_{\to}, c_{\times}, c_{R})) & = & ((\texttt{fix}\, f{:}\forall\alpha{:}\mathsf{Type}.R(\alpha) \to R(c^*[\alpha]{:}\kappa). \\
& & \quad \Lambda\alpha{:}\mathsf{Type}. \\
& & \quad\quad \lambda x_\alpha{:}R(\alpha). \\
& & \quad\quad\quad \texttt{typecase}[\delta.R(c^*[\alpha]{:}\kappa)]\ x_\alpha \\
& & \quad\quad\quad\quad R_{int} \Rightarrow \Re(c_{int}) \\
& & \quad\quad\quad\quad R_{\to}(x_\beta, x_\gamma)\ \mathsf{as}\ (\beta \to \gamma) \Rightarrow \\
& & \quad\quad\quad\quad\quad \Re(c_{\to})[\beta]x_\beta\ [\gamma]x_\gamma \\
& & \quad\quad\quad\quad\quad\quad [c^*[\beta]](f[\beta]x_\beta)\ [c^*[\gamma]](f[\gamma]x_\gamma) \\
& & \quad\quad\quad\quad R_{\times}(x_\beta, x_\gamma)\ \mathsf{as}\ (\beta \times \gamma) \Rightarrow \\
& & \quad\quad\quad\quad\quad \Re(c_{\times})[\beta]x_\beta\ [\gamma]x_\gamma \\
& & \quad\quad\quad\quad\quad\quad [c^*[\beta]](f[\beta]x_\beta)\ [c^*[\gamma]](f[\gamma]x_\gamma) \\
& & \quad\quad\quad\quad R_{R}(x_\beta)\ \mathsf{as}\ R(\beta) \Rightarrow \\
& & \quad\quad\quad\quad\quad \Re(c_{R})[\beta]x_\beta[c^*[\beta]](f[\beta]x_\beta) \\
& & \quad [\tau]\Re(\tau)) \\
& & \text{where } c^*[\tau'] = \texttt{Typerec}\ \tau'(c_{int}, c_{\to}, c_{\times}, c_{R})
\end{array}
$$