### Citation

### Published Version
http://doi.acm.org/10.1145/289423.289459

### Citable link
http://nrs.harvard.edu/urn-3:HUL.InstRepos:3303080

### Terms of Use
This article was downloaded from Harvard University’s DASH repository, and is made available under the terms and conditions applicable to Other Posted Material, as set forth at http://nrs.harvard.edu/urn-3:HUL.InstRepos:dash.current.terms-of-use#LAA
Intensional Polymorphism in Type-Erasure Semantics

Karl Crary        Stephanie Weirich        Greg Morrisett

Cornell University

Abstract

Intensional polymorphism, the ability to dispatch to different routines based on types at run time, enables a variety of advanced implementation techniques for polymorphic languages, including tag-free garbage collection, unboxed function arguments, polymorphic marshalling, and flattened data structures. To date, languages that support intensional polymorphism have required a type-passing (as opposed to type-erasure) interpretation where types are constructed and passed to polymorphic functions at run time. Unfortunately, type-passing suffers from a number of drawbacks; it requires duplication of 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 [12, 10, 16, 1, 22, 24, 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 $\alpha$ where $\alpha$ 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 [10, 16, 27, 18, 24]. Such type information is used in two ways: behind the scenes, typically by tag-free garbage collectors [27], and explicitly in program code, for a variety of purposes such as efficient data representation and marshalling [16, 10]. 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 $\lambda^{ML}$, that supports the first-class intensional analysis of types (following earlier work by Constable [2, 3]). The $\lambda^{ML}$ language and its derivatives were then used extensively in the high-performance ML compilers TIL/ML [26, 19] and FLINT [25]. The primary novelty of $\lambda^{ML}$ 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.

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 $\lambda^{ML}$ requires that type information always be constructed and passed to polymorphic functions. This 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 [17, 18] a central device is a formal heap in which data is stored; in a type-erasure framework one such heap

\footnote{“Intensional” since types are analyzed by the structure of their names, rather than by what terms they contain. This is critical for practicality.}
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 [20] 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. [14] to the type-erasure system of Morrisett et al. [20]). 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 [14].

In this paper we propose a typed calculus, called $\lambda_R$, that ameliorates the problems of type passing without sacrificing intensional type analysis. If run-time type dispatch is to be 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 $\tau$ 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 $\tau$.

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 4. Finally, our approach enables the choice not to pass representations. In turn, this allows us to eliminate the overhead of constructing and passing representations of types where it is not necessary.

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 [21] or through existential types [15]. However, when all types are passed and may be analyzed (as in $\lambda^{\mu\nu}$), the identity of types cannot be hidden and consequently abstraction is impossible. In contrast, in $\lambda_R$ a 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, this implements an abstract type (in this particular example, a useless abstract type), 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 $\lambda^{\mu\nu}$ and focus on their differences. The principal difference is the restriction of type analysis to those types for which representations are provided. This does not diminish the expressiveness of our calculus; $\lambda^{\mu\nu}$ may be translated in a straightforward syntax-directed manner into $\lambda_R$.

Moreover, the $\lambda_R$ calculus incorporates an additional improvement in expressiveness over $\lambda^{\mu\nu}$ that is independent of explicit type passing: In $\lambda^{\mu\nu}$, 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 with 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 $\lambda^{\mu\nu}$ 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.

### 1.2 Overview

The remainder of this paper is organized as follows: In Section 2 we review the $\lambda^{\mu\nu}$ 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. In Section 4, we discuss the simplification of polymorphic closure conversion by explicit type passing. We end with related work and conclusions in Sections 5 and 6. In the appendices we relate our typed semantics to an untyped one through type erasure (Appendix A), discuss the analysis of quantified types (Appendix B), and provide the complete dynamic and static semantics.
2 Intensional Type Analysis

Suppose we wanted to store an array of booleans. 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” 32 booleans into one word and use bit manipulations to retrieve the correct value. So, to subscript from a boolean array, we would use the following function:

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

This is fine when we know a given array contains boolean values, but we would like code that is polymorphic over the type of array to be able to use this mechanism. In essence, we need to 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 need an associated subscript operation, `packedsub`, so that when the array is of booleans, `bitsub` is called. This can be done with intensional type analysis, where in both cases an argument type is examined with a `typecase` form:

```plaintext
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 $\lambda^{\mu}$ calculus

To formalize the tools of intensional type analysis, we begin by summarizing Harper and Morrisett’s $\lambda^{\mu}$ calculus [10]. The $\lambda^{\mu}$ calculus provides these tools in a form that is relatively simple, but already quite powerful.

The syntax of $\lambda^{\mu}$ appears below (modified slightly for presentation). The backbone of $\lambda^{\mu}$ is a predicative variant of Girard’s $F_\omega$ [8, 7] in which the quantified type $\forall:\alpha:K.\sigma$ 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.

```plaintext
(kinds)  κ :: = Type | κ₁ → κ₂
(con’s)  e :: = α | int | c₁ → c₂ | c₁ × c₂ | λα:κ.e | c₁ c₂ |
          | Typerec c [c₁, c₂].c
(types)  σ :: = c | α → σ | σ → c | σ × c | |∀α:κ.σ |
(terms)  e :: = x | i | λα:κ.e | fix f:σ.v | e₁e₂ | (e₁, e₂) | π₁e | π₂e |
          | λα:κ.e | e | typecase [α, σ].c
          | int => e_int
          | β → γ => e_β
          | β × γ => e_γ
(values) v :: = x | i | λα:κ.e | fix f:σ.v |
          | (e₁, e₂) | Λα:κ.e
```

Occasionally, for brevity, we will write `typecase` terms as `typecase[α, σ].c` (e_int, β_γ, e_, β_γ, e_γ).

As an example of the use of type analysis in $\lambda^{\mu}$, consider the function `tostring`, presented in Figure 1. With the addition of another base type, `string`, to the language, 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:

```plaintext
tostring [(int → int) × int] (λx:int.x + 1, 3)
```

which returns

“(function, 3)”.

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

```plaintext
fix tostring : (∀α:Type. α → string).
  Λα:Type.
  typecase[δβ → string] α of
  int => int2string
  string => λobj:string.obj
  β → γ =>
  λobj:(β → γ)."function"
  β × γ =>
  λobj:(β × γ).
  "<"*(tostring[β](π₁ obj))"*
  "*"*(tostring[γ](π₂ obj))"*=>*
```

Figure 1: The function `tostring`

The `typecase` form also has a type annotation to make it possible to type check it without type inference; the annotation $[α, σ]$ indicates that given a type argument $τ$, the `typecase` computes a value with type $σ[τ/α]$ (where this denotes the capture-avoiding substitution of $τ$ 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 is somewhat restrictive):

\[
\frac{c : \text{Type} \quad \alpha : \text{Type} \vdash \sigma \text{ type}}{\Gamma \vdash \text{typecase}[\alpha, \sigma](e) : \sigma[c/\alpha]} \\
\frac{c : \text{Type} \quad \beta : \text{Type}, \gamma : \text{Type} \vdash e_{\ast} : \sigma[\beta \rightarrow \gamma/\alpha]}{\Gamma, \beta : \text{Type}, \gamma : \text{Type} \vdash e_{\ast} : \sigma[\beta \times \gamma/\alpha]} \\
\frac{e_{\ast} \rightarrow e_{\ast}}{\Gamma \vdash \text{typecase}[\alpha, \sigma](e) : \sigma[c/\alpha]} \\
\frac{\beta \rightarrow \gamma \Rightarrow e_{\ast} \quad \beta \times \gamma \Rightarrow e_{\ast}}{\Gamma \vdash \text{typecase}[\alpha, \sigma](e) : \sigma[c/\alpha]}
\]

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 this appear in Harper and Morrisett [10], 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 \( c_{\ast} \) 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 these components.

\[
\text{Typerec}(c_1 \times c_2) =
\begin{cases}
    c_{\times} & c_1 \times c_2 \\
    \text{Typerec}(c_1(c_{\ast_1}, c_{\ast_2}, c_{\times})) & \\
    \text{Typerec}(c_2(c_{\ast_1}, c_{\ast_2}, c_{\times})) &
\end{cases}
\]

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:

\[
\frac{\Gamma \vdash c : \text{Type}}{\Gamma \vdash \text{Typerec}(c) : \kappa} \\
\frac{\Gamma \vdash c_{\ast} : \text{Type} \quad \Gamma \vdash c_{\times} : \text{Type} \quad \Gamma \vdash c_{\times} : \text{Type}}{\Gamma \vdash \text{Typerec}(c) : \kappa}
\]

\[3\] The \( \lambda_R \) calculus

Figure 2 presents the syntax of \( \lambda_R \), which we describe in detail in the following section.

![Figure 2: Syntax of \( \lambda_R \)](image)

### 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, instead of a type. For example, if the argument \( e \) is of the form \( R_{\ast}(e_1, e_2) \), the arrow branch \( e_{\ast} \) is taken. The type variables \( \beta \) and \( \gamma \) 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 is reflected in the following rule of the operational semantics:

\[
\text{typecase}[\beta, \gamma](R_{\ast}(e_1, e_2)) = (e_1, e_2, \beta \gamma x y, e_{\ast}, \ldots)
\]

The operational semantics for \( \lambda_R \) is given by the operational rules formalized in Figure 4.

Figure 2 presents the syntax of \( \lambda_R \), which we describe in detail in the following section.
accordingly. For example, the formation rule for the representation of function types is

$$\Gamma \vdash e_1 : R(\tau_1) \quad \Gamma \vdash e_2 : R(\tau_2) \quad (\text{rep}_\rightarrow)$$

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

As an example of the use of $\lambda_R$, the tostring function from the previous section can be transcribed into R-types. We also extend typecase.

The typing rules of $\lambda_R$’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. We separate the formation rule into situations where typecase gives us new information, such as when the argument is of type $R(\alpha)$, and when it does not, such as when the argument is of type $R(\tau_1 \rightarrow \tau_2)$. In the inference rule for type checking a typecase term, when the argument is of type $R(\alpha)$, we refine types containing $\alpha$ to reflect the gain in information, as shown below. For simplicity, only some of the rule is included (the complete rule can be found in Figure 8):

$$\begin{align*}
\Gamma, \alpha: \text{Type}, \Gamma' &\vdash e : R(\alpha) \\
\Gamma, \Gamma'[\int/\alpha] &\vdash e_{\text{int}}[\int/\alpha] : c[\int, \Gamma / \alpha, \delta] \\
\Gamma, \beta: \text{Type}, \gamma: \text{Type}, \Gamma'[\beta \rightarrow \gamma/\alpha], x: R(\beta), \\
y: R(\gamma) &\vdash e_{\rightarrow}[\beta \rightarrow \gamma/\alpha] : c[\beta \rightarrow \gamma, \beta \rightarrow \gamma/\alpha, \delta] \\
\end{align*}$$

The reason we could not do this in $\lambda^{ML}_R$ is that we need the type of $\text{obj}$ to change based upon which branch of the typecase is selected. All we know in the product branch is that $\text{obj}$ is of type $\alpha$, not a tuple. In order to project from it in the recursive calls, we need to update the type of $\text{obj}$ to reflect the fact that we know that $\alpha$ is $\beta \times \gamma$ in the product branch.

With the right enhancement to the static semantics this is possible. We have held off the discussion of the $\lambda^{ML}_R$’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. We separate the formation rule into situations where typecase gives us new information, such as when the argument is of type $R(\alpha)$, and when it does not, such as when the argument is of type $R(\tau_1 \rightarrow \tau_2)$. In the inference rule for type checking a typecase term, when the argument is of type $R(\alpha)$, we refine types containing $\alpha$ to reflect the gain in information, as shown below. For simplicity, only some of the rule is included (the complete rule can be found in Figure 8):

$$\begin{align*}
\Gamma, \alpha: \text{Type}, \Gamma' &\vdash e : R(\alpha) \\
\Gamma, \Gamma'[\int/\alpha] &\vdash e_{\text{int}}[\int/\alpha] : c[\int, \Gamma / \alpha, \delta] \\
\Gamma, \beta: \text{Type}, \gamma: \text{Type}, \Gamma'[\beta \rightarrow \gamma/\alpha], x: R(\beta), \\
y: R(\gamma) &\vdash e_{\rightarrow}[\beta \rightarrow \gamma/\alpha] : c[\beta \rightarrow \gamma, \beta \rightarrow \gamma/\alpha, \delta] \\
\end{align*}$$

For example, to typecheck the $e_{\rightarrow}$ branch, we substitute $\beta \times \gamma$ for $\alpha$ everywhere, including the surrounding context. Consequently the types of the variables bound in the context will be refined by that substitution. Because $\lambda^{ML}_R$ 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 we know more about the argument because of the singularity of the R-types, we can deduce statically which branch of the typecase will be taken. Therefore we do not need to typecheck the other branches at all, leading to a much simpler rule. For example, if we know the argument is of type $R(\tau_1 \rightarrow \tau_2)$, we only need to examine the $e_{\rightarrow}$ branch, as in the rule:

$$\begin{align*}
\Gamma &\vdash e : R(\tau_1 \rightarrow \tau_2) \\
\Gamma, x: R(\tau_1), y: R(\tau_2) &\vdash e_{\rightarrow}[\tau_1, \tau_2/\beta, \gamma] : c[\tau_1 \rightarrow \tau_2/\delta] \\
\Gamma &\vdash \text{typecase}_{\delta, c} e (e_{\text{int}}, \beta \gamma x y e_{\rightarrow}, \ldots) : c[\tau_1 \rightarrow \tau_2/\delta] \\
\end{align*}$$

3.3 Impredicativity

A final minor difference between $\lambda_R$ and $\lambda^{ML}_R$ is that we have chosen to make $\lambda_R$ impredicative. We do this because our interest in $\lambda_R$ is for compilation where it is difficult in some cases (such as typed closure conversion, 

---

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 by a local type definition, rather than by substitution.
### 3.4 Properties of the Formal Semantics

Formally, the static semantics of $\lambda_R$ consists of a collection of rules for deriving judgments of the forms shown in Figure 3. In these judgments, $\Gamma$ is a unified type and kind context, mapping either constructor variables $(\alpha, \beta, \ldots)$ to kinds, or term variables $(x, y, \ldots)$ to types. The full formal static and operational semantics of $\lambda_R$ appear in Figures 4–9, and from them we can prove several useful properties about $\lambda_R$.

First, we would like to prove the decidability of $\lambda_R$ type-checking. The difficult part of the that is equivalence checking for constructors. Based upon the equivalence rules in Figure 7 we can define a notion of constructor reduction to a normal form in an obvious manner. This reduction relation can be proved to be strongly normalizing and confluent (in a manner similar to Morrisett [16]) 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 $\lambda_R$.

Next, we would like to show that the static semantics guarantees safety; that is, if a term type checks, 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 $\emptyset \vdash e : \tau$ and $e \mapsto^* e'$ then $e'$ is not stuck.

The proof of this theorem is standard, relying on the usual progress, subject reduction and substitution lemmas.

### 4 Polymorphic Typed Closure Conversion

As a final example, we consider typed closure conversion in a $\lambda_R$-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 $\lambda$-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 [14] and type-erasure [20] closure conversion. An existential type is used to hold the type of the closure’s environment abstract, so a closure for a $\tau_1 \rightarrow \tau_2$ function is given the type $\exists\alpha.((\tau_1 \times \alpha) \rightarrow \tau_2) \times \alpha$.

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 not only free value variables, but also free type variables, and closed code must abstract these as well. Closures must then provide somehow for applying such code to the appropriate type variables. In a type-erasure setting, application to type arguments has no run-time effect, so the partial application of code to the appropriate type variables may be performed when closures are created. Consequently, the possibility of free type variables does not figure into the type of a closure, and so closures have the same type ($\exists\alpha.((\tau_1 \times \alpha) \rightarrow \tau_2) \times \alpha$) 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. [14] for detailed explanations of why). The former requires the use of abstract kinds and the latter requires the use of translucent types [9]. This results in a closure having the considerably more complicated type (again, see Minamide et al. [14] for a
formalization of the necessary type theory):

\[ \exists k_{\text{tenv}}, \text{Kind}, \exists \alpha_{\text{tenv}}, \text{Type}, \exists \beta_{\text{tenv}}, k_{\text{tenv}}. \\
(V\forall \gamma: k_{\text{tenv}} \Rightarrow \beta_{\text{tenv}}, (\tau_1 \times \alpha_{\text{tenv}}) \Rightarrow \tau_2) \times \alpha_{\text{tenv}} \]

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

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 is rewritten in closed form as:

\[ f' : \forall \alpha. (\tau_1 \times R(\alpha)) \Rightarrow \tau_2 \]

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

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

Finally, a closure is created:

\[ f''' = \text{pack} (f'', x_\alpha) \text{ as } \exists \beta. ((\tau_1 \times \beta) \Rightarrow \tau_2) \times \beta \text{ hiding } R(\alpha) \]

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 \( \alpha \). 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 (but see Appendix B).

5 Related Work

Closely related to our work is the work of Minamide on lifting of type parameters for tag-free garbage collection [13]. 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 or first-class polymorphism. Minamide’s system also makes a distinction between type representations (which he calls evidence, following Jones [11]) and ordinary terms, while \( \lambda \) type representations are fully first-class.

The issue of type parameter lifting is an important one for compilers based on \( \lambda \). The construction of type representations at run time would likely lead to significant cost and should in practice be lifted out to compile time whenever possible. (Unfortunately, in the presence of polymorphic recursion, which \( \lambda \) supports, it cannot always be possible.) Mechanisms for lifting such lifting have been developed by Minamide (in the work discussed above) and by Saha and Shao [23].

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 \( \lambda^{t/c} \). Like \( \lambda^{t/c} \), 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 \( \lambda^{t/c} \) and \( \lambda \) 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.

6 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. [20] 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 fully 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 [26, 19] and FLINT [25], and has made it infeasible to use intensional type analysis in the end-to-end typed compiler TALC [20]. This 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 \( \lambda \) into lower-
level typed intermediate languages such as typed assembly language [20]. 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 \texttt{typedef} constructor. Another issue to explore is better mechanisms for analysis of quantified types (some initial ideas appear in Appendix B), and whether such mechanisms are merited in practice.

Another important question is whether a parametricity theorem like that of Reynolds [21] can be shown for \( \lambda_R \). Polymorphism is clearly non-parametric in \( \lambda_{\mu} \), 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 \( \lambda_{\mu} \); it would be interesting to explore how that intuition may be formalized.

References


A Untyped Calculus

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. To do this we exhibit an untyped language, $\lambda R^0$, 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 $e_1^0 \mapsto^* e_2^0$.
2. If $\not\Gamma \vdash e : \tau$ and $e_1^0 \mapsto^* u$ then there exists $e_2$ such that $e_1 \mapsto^* e_2$ and $e_2^0 = u$.

From this theorem and type safety for $\lambda R$ it follows that our untyped semantics is safe.

**Corollary A.2** If $\not\Gamma \vdash e : \tau$ and $e^0 \mapsto^* u$ then $u$ is not stuck.

### A.1 Syntax of Untyped Calculus

\[
\begin{align*}
(\text{terms}) & \quad u := \ x \mid \lambda x. u \mid \text{fix} \ f. w \mid u_1 u_2 \\
& \quad \mid (u_1, u_2) \mid \pi_1 u \mid \pi_2 u \\
& \quad \mid R_{\text{int}}(u_1, u_2) \mid R_\delta(u_1, u_2) \mid R_\tau(u) \mid R_\nu \mid R_\gamma \\
(\text{typecase}) & \quad \text{of} \\
& \quad R_{\text{int}} \Rightarrow u_{\text{int}} \\
& \quad R_\delta(x, y) \Rightarrow u_\delta \\
& \quad R_\tau(x, y) \Rightarrow u_\tau \\
& \quad R_\gamma(x) \Rightarrow u_\gamma \\
& \quad R_\nu \Rightarrow u_\nu \\
& \quad R_\gamma \Rightarrow u_\gamma \\
(\text{values}) & \quad w := \ x \mid \lambda x. u \mid \text{fix} \ f. w \\
& \quad \mid R_{\text{int}}(u_1, u_2) \mid R_\delta(w_1, w_2) \mid R_\tau(w_1, w_2) \\
& \quad \mid R_\nu \mid R_\gamma \\
\end{align*}
\]

### A.2 Type Erasure

\[
\begin{align*}
\begin{array}{ll}
x^0 & = x \\
i^0 & = i \\
(e_1, e_2)^0 & = (e_1^0, e_2^0) \\
(\pi_1 e)^0 & = \pi_1 e^0 \\
(\lambda x : e)^0 & = \lambda x. e^0 \\
(\lambda x : k : v)^0 & = v^0 \\
(\text{fix} \ f. c. v)^0 & = \text{fix} f. v^0 \\
(e_1 e_2)^0 & = e_1^0 e_2^0 \\
e[c]^0 & = e^0 \\
\text{pack} e \text{ as } \text{ hiding } c^0 & = e^0 \\
\text{unpack} (\alpha, x) & = e_1 \text{ in } e_2^0 = (\lambda x. e_2^0) e_1^0 \\
R_{\text{int}} & = R_{\text{int}} \\
R_\delta(e_1, e_2)^0 & = R_\delta(e_1^0, e_2^0) \\
R_\tau(e_1, e_2)^0 & = R_\tau(e_1^0, e_2^0) \\
R_\gamma(e_1)^0 & = R_\gamma(e_1^0) \\
R_\nu & = R_\nu \\
R_\gamma & = R_\gamma \\
\end{array}
\]

\[(\text{typecase}[\delta, c]. e). \text{ of} \quad \text{of} \\
\begin{array}{ll}
R_{\text{int}} & \Rightarrow e_{\text{int}} \\
R_\delta & \Rightarrow e_{\text{int}} \\
R_\delta(x, y) \Rightarrow & R_\delta(x, y) \Rightarrow e_{\text{int}} \\
(\beta \rightarrow \gamma) & \Rightarrow e_{\text{int}} \\
R_\delta(x, y) & \Rightarrow e_{\text{int}} \\
(\beta \times \gamma) & \Rightarrow e_{\text{int}} \\
R_\delta(x) & \Rightarrow e_{\text{int}} \\
R_\gamma & \Rightarrow e_{\text{int}} \\
R_\nu & \Rightarrow e_{\text{int}} \\
R_\gamma & \Rightarrow e_{\text{int}} \\
\end{array}
\]

### A.3 Operational Semantics of $\lambda R^0$

\[
\begin{align*}
(\lambda x : c. u)[w/x] & \Rightarrow u[w/x] \\
(\text{fix} f : c. w)[w'] & \Rightarrow (w[\text{fix} f : c. w/f])[w'] \\
\pi_1(w_1, w_2) & \Rightarrow w_1 \\
\pi_2(w_1, w_2) & \Rightarrow w_2 \\
\text{typecase} R_{\text{int}}(u_{\text{int}}, x y. u x y) & \Rightarrow u_{\text{int}} \\
x y. u & \Rightarrow x y. u x y \\
\end{align*}
\]
types at any other kind $\kappa$ would be built by an analogous constructor $R_{\forall \kappa}$ or $R_{\exists \kappa}$. Note that this requires an infinite collection of such constructors, two for each kind.

Analysis of types is also straightforward if quantification is restricted to Type, or to any finite set of kinds. In that case, the appropriate branches can be added to the Typerec and typecase operations, and each branch would return the appropriate representation function discussed above. General quantification could be handled, at the expense of additional complexity, by adding kind variables, term representations of kinds, and a Kindrec facility for analyzing such representations. With such additions, the quantifier branches would return a kind representation and the appropriate representation function.

However, with such a mechanism in place, the amount of useful analysis that can be performed is still quite limited. Type analyzing code may apply the representation function to an argument and analyze its output, but it could not analyze the function itself. This makes it impossible, for example, to print quantified types. The modal type theory of Despeyroux et al. [4] is intended for precisely this sort of application; it provides mechanisms for primitive recursion on higher-order syntax and might provide a solution to this problem.

Also, allowing any analysis of quantified types by Typerec sacrifices strong normalizations of type expressions. (An isomorphism between Type and Type $\rightarrow$ Type can be built, permitting the encoding of the untyped lambda calculus.) A modal type discipline holds some promise of a solution to this problem as well.

### B Analysis of Quantified Types

The analysis of quantified types in $\lambda_r$, as presented in this paper, is limited to the outermost operator; that is, a Typerec or typecase will determine that a type is a $\forall$ or $\exists$ type, but will not provide any information about the body of the type. The principal reasons for this decision are that support for full analysis of quantified types detracts from the elegance of the type theory, and that experience from the TIL/ML compiler suggests that such support may not be needed in practice. (In fact, the TIL/ML compiler uses a variant of $\lambda^{ML}$ that handles quantified types in a manner relatively similar to $\lambda_r$.) In this section we briefly explore what would be required to extend $\lambda_r$ to support full analysis of quantified types.

Adding term-level representations for quantified types is straightforward. The representation for $\forall \alpha : \kappa. c$ must carry information expressing how to produce the appropriate $c$ given an appropriate $\alpha$ of kind $\kappa$. For example, a member of $R(\forall \alpha : \text{Type}. c)$ would be built by a constructor $R_{\forall \text{Type}}$ that takes a function with type $\forall \alpha : \text{Type}. R(\alpha) \rightarrow R(c)$. Representations of polymorphic
Figure 4: Operational Semantics for $\lambda_R$

$$
\begin{align*}
&\text{(pack } e \text{ as } \exists x. e) \mapsto e & \text{(unpack } \alpha, x) = \text{ pack } \nu \text{ as } \exists \beta. \pi \text{ hiding } e_2 \mapsto e_2[v, \alpha, x] \\
&\text{typecase}[\delta, c] R_\text{int} (e_\text{int}, \beta_\gamma y, e_\gamma, \beta x, e_x, \beta e_x, e_x) \mapsto e_\text{int} \\
&\text{typecase}[\delta, c] (R_e(e')) (e_\text{int}, \beta_\gamma y, e_\gamma, \beta x, e_x, \beta e_x, e_x) \mapsto e_\text{int} \\
&\text{typecase}[\delta, c] (R_\gamma(e')) (e_\text{int}, \beta_\gamma y, e_\gamma, \beta x) \mapsto e_\text{int} \\
\end{align*}
$$
Figure 6: Constructor formation

Figure 7: Constructor equivalence (selected rules)
\[
\begin{align*}
\Gamma \vdash e : c \\
\Gamma \vdash i : \text{int} \quad (\text{int}) \quad \Gamma \vdash x : c \quad (\Gamma(x) = c) \quad (\text{var}) \quad \Gamma \vdash e_1 : c_1 \quad \Gamma \vdash e_2 : c_2 \quad (\text{pair}) \quad \Gamma \vdash e : c_1 \times c_2 \quad (\text{proj}_i) \\
\Gamma \vdash e_1 : c' \quad \Gamma \vdash e_2 : c' \quad (\text{app}) \quad \Gamma \vdash e : c' \quad (\text{tapp}) \quad \Gamma \vdash e[c'/c] : c' \quad (\text{fix}) \\
\Gamma, c_1 : \text{Type} \quad \Gamma \vdash e_1 : c' \quad \Gamma \vdash e_2 : c' \quad \Gamma \vdash e : c' \quad (\text{pack}) \quad \Gamma \vdash \text{pack} e \text{ as } \exists \alpha : k . c' \quad (\text{equiv}) \\
\Gamma \vdash \text{R} \alpha \vdash R \int \quad (\text{rep}_{\text{int}}) \quad \Gamma \vdash e_1 : R(c_1) \quad \Gamma \vdash e_2 : R(c_2) \quad (\text{rep}_{\text{r}}) \quad \Gamma \vdash \text{R} \alpha \text{ select } e_1, e_2 : \text{R} (c_1 \rightarrow c_2) \quad (\text{rep}_{\text{r}}) \\
\Gamma \vdash e : R(c) \quad (\text{rep}) \quad \Gamma \vdash e_1 : R(c_1) \quad \Gamma \vdash e_2 : R(c_2) \quad (\text{rep}_{\text{r}}) \quad \Gamma \vdash R_{\alpha} (e_1, e_2) : R (c_1 \times c_2) \quad (\text{rep}_{\text{r}}) \\
\Gamma \vdash \text{R} \alpha \text{ : Type} \quad \Gamma \vdash \text{R} (\forall \alpha : k . c) : \text{R} (\forall \alpha : k . c) \quad (\text{rep}_{\text{r}}) \quad \Gamma \vdash \text{R} \alpha \text{ : Type} \quad \Gamma \vdash \text{R} (\exists \alpha : k . c) : \text{Type} \quad (\text{rep}_{\text{r}}) \\
\Gamma \vdash e : e' \quad \Gamma \vdash c' : \kappa \quad (\text{equiv})
\end{align*}
\]

Figure 8: Term formation (except typecase)
Figure 9: Term Formation (typecase)