Higher-Order Intensional Type Analysis in Type-Erasure Semantics

Stephanie Weirich
sweirich@cis.upenn.edu

Department of Computer and Information Science
University of Pennsylvania

Abstract

Higher-order intensional type analysis is a way of defining type-indexed operations, such as map, fold and zip, based on run-time type information. However, languages supporting this facility are naturally defined with a type-passing semantics, which suffers from a number of drawbacks. This paper, describes how to recast higher-order intensional type analysis in a type-erasure semantics. The resulting language is simple and easy to implement—we present a prototype implementation of the necessary machinery as a small Haskell library.

1 Polytypic programming

Some functions are naturally defined by the type structure of their arguments. For example, a *polytypic* pretty printer can format any data structure by using type information to decompose it into basic parts. Without such a mechanism, one must write separate pretty printers for all data types and constantly update them as data types evolve. Polytypic programming simplifies the maintenance of software by allowing functions to automatically adapt to changes in the representation of data. Other classic examples of polytypic operations include reductions, comparison functions and mapping functions. The theory behind such operations has been developed in a variety of frameworks [1, 2, 5, 7, 11, 12, 13, 14, 21, 24, 26, 28].

While many of these frameworks generate polytypic operations at compile time (through a source-to-source translation determined by static type information), *higher-order intensional type analysis* [31] defines polytypic operations with *run-time* type information. Run-time type analysis has two advantages over static forms of polytypism: First, run-time analysis may index polytypic operations by types that are not known at compile time, allowing the language to support separate compilation, dynamic loading and polymorphic recursion. Second, run-time analysis may index polytypic operations by universal and existential types. Because these quantified types hide information, the semantics of the programming language must provide type information at run time to define most operations over these types.

Draft of July 2003

Run-time type analysis is naturally defined by a type-passing semantics because types play an essential role in the execution of programs. However, there are several significant reasons to prefer a semantics where types are erased prior to execution:

- A type-passing semantics *always* constructs and passes type information to polymorphic functions. It cannot support abstract data types because the identity of any type may be determined at run time. Furthermore, parametricity theorems [20, 27] about polymorphic terms are not valid with this semantics.
- Because both terms and type constructors describe run-time behavior, type passing results in considerable complexity in the semantics of languages that precisely describe execution. For example, a language that makes memory allocation explicit [16, 17] uses a formal heap to model how data is stored; with run-time types it is necessary to add a second heap (and all the attendant machinery) for type data.
- Operators that implement type analysis in a type-erasure semantics are easier to incorporate with existing languages (such as Haskell and ML) that already have this form of semantics. Extending these languages with this form of type analysis does not require global changes to their implementations. In fact, for some languages it is possible to define type analysis operators with library routines written in that language. For example, Weirich [30] shows how to encode first-order run-time type analysis in F_ω [9] and Cheney and Hinze [3] implement the same capabilities in the Haskell language [19].

In *first-order* intensional type analysis, types such as *int* and *bool* $\times$ *string* are the subject of analysis—an operator called *typerec* computes a catamorphism over the structure of run-time types. The idea behind *higher-order* intensional analysis is that the structure of parameterized types (i.e. higher-order type constructors) is examined. In this framework, *typerec* acts like an environment-based interpreter of the type language during execution. Higher-order analysis can define more polytypic operations than first order analysis. For example, a polytypic function that counts the

		Type analysis	Semantics
λ_R	[6]	First-order	Type-erasure
LH	[31]	Higher-order	Type-passing
LHR		Higher-order	Type-erasure

Figure 1: Language comparison

number of values of type α in a parameterized data structure of type $\tau\alpha$ must analyze the type constructor τ. Many of the most important examples of polytypic programming are only definable by higher-order analysis, including maps, zips, folds and reductions.

Crary, Weirich and Morrisett [6] (CWM) describe how to support first-order intensional type analysis in a language with a type-erasure semantics. In their language λ_R, *typerec* examines terms that represent types instead of analyzing types. In a type-erasure version of higher-order analysis, *typerec* should examine term representations of higher-order type constructors. However, while CWM define representations of higher-order type constructors in λ_R, these representations cannot be used for higher-order analysis. For technical reasons discussed in Section 3, we cannot define a term that operates over these type constructor representations in the same way as the type-passing *typerec* term operates over type constructors. These difficulties prohibit an easy definition of a type-erasure language that may define higher-order polytypic operations.

In this paper, we show how to reconcile higher-order analysis with type erasure. Our specific contributions include:

- A language, called LHR, that supports higher-order intensional type analysis in a type-erasure semantics. Surprisingly, in some respects LHR is a simpler calculus than the type-passing version of higher-order type analysis.
- A translation between the type-passing version of higher-order intensional type analysis and LHR, with a proof of correctness.
- A prototype implementation of LHR as a Haskell library that is simple, easy to use and specialized to the Haskell type system, allowing polytypic functions to operate over built-in Haskell datatypes.

The structure of this paper is as follows. Section 2 reviews higher-order intensional type analysis (formalized with the language LH) and Section 3 discusses the problems with defining a type-erasure version of this language. In Section 4 we present the type-erasure language called LHR. We describe the translation between LH and LHR in Section 5. Section 6 describes the prototype implementation of LHR as a Haskell library. In Section 7 we discuss extensions of this translation, and in Section 8 we present related work and conclude. Appendix A contains the proof of correctness of the translation.

2 LH: Higher-order analysis with type-passing

The LH language (Figure 2) is a lightweight characterization of higher-order intensional type analysis that captures the core ideas of the language of Weirich [31]. It is a call-by-name variant of the Girard-Reynolds polymorphic lambda calculus [10, 9, 20] plus the *typerec* term to define polytypic operations.[1] The choice of call-by-value or call-by-name is not significant, and call-by-name slightly simplifies the presentation. Also for simplicity, the formal language contains only integers, functions, and polymorphic terms, although we will include additional forms (such as products, sums, and term and type recursion, with their usual semantics) in the examples. The behavior of *typerec* on these new type forms is analogous to that for integers, functions and polymorphic types.

(kinds)	$\kappa ::= \star \mid \kappa_1 \to \kappa_2$
(operators)	$\oplus ::= \mathit{int} \mid \to \mid \forall_\star$
(type constructors)	$\tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 \mid \oplus$
(types)	$\sigma ::= \tau \mid \mathit{int} \mid \sigma_1 \to \sigma_2 \mid \forall\alpha{:}\kappa.\sigma$
(terms)	$e ::= i \mid x \mid \lambda x{:}\sigma.e \mid e_1e_2$
	$\mid \Lambda\alpha{:}\kappa.e \mid e[\tau]$
	$\mid \mathit{typerec}\,[\Delta,\eta,\rho][\tau']\langle\tau : \kappa\rangle\ \mathit{of}\ \ \theta$
(typerec branches)	$\theta ::= \emptyset \mid \theta\{\oplus \Rightarrow e\}$
(term environment)	$\eta ::= \emptyset \mid \eta\{\alpha \Rightarrow e\}$
(tycon environment)	$\rho ::= \emptyset \mid \rho\{\alpha \Rightarrow \tau\}$
(tycon context)	$\Delta ::= \emptyset \mid \Delta\{\alpha \Rightarrow \kappa\}$
(term context)	$\Gamma ::= \emptyset \mid \Gamma\{x \Rightarrow \sigma\}$
(operator signature)	$\Sigma ::= \{\ \mathit{int} \Rightarrow \star,$
	$\to\ \Rightarrow \star \to \star \to \star,$
	$\forall_\star\ \Rightarrow (\star \to \star) \to \star\}$

Figure 2: Syntax of LH

Types, σ, which describe terms, are separated from type constructors, τ, although we often call type constructors of base kind, $\star$, types. The operators, $\oplus$, are a set of constants of the type constructor language. These constants correspond to the various forms of types: for example, the constant $\to$ applied to τ_1 and τ_2 is equivalent to the function type $\tau_1 \to \tau_2$, and $\forall_\star\tau$ is equivalent to the type $\forall\alpha{:}\star.\tau\alpha$.[2] The signature, Σ, is a fixed finite map that describes the kinds of the operators. We use the notation $\Sigma(\oplus)$ to refer to the kind of the operator $\oplus$. The language includes several other finite maps, such as ρ, η, θ, etc. We write the empty map as $\emptyset$, add a new binding to ρ with $\rho\{\alpha \Rightarrow \tau\}$ (defined only when $\alpha \notin \mathit{Dom}(\rho)$) and retrieve a binding with $\rho(\alpha)$ (defined only when $\alpha \in \mathit{Dom}(\rho)$). We also use the notation $\rho(\tau)$ to substitute in τ for all variables bound in ρ. The notation for the other maps is analogous.

The term $\mathit{typerec}\,[\Delta',\eta,\rho][\tau']\langle\tau : \kappa\rangle\ \mathit{of}\ \ \theta$ defines polytypic operations. Essentially, it behaves like an interpreter of the type constructor language, translating the type constructor τ (of kind κ) to an element of the term language using the branches θ for the interpretation of operators and the environment η for the interpretation of type variables. The *typerec* term is the binding occurrence for the variables that might appear in τ at run-time—those that have a definition in the environment η. The context Δ' describes the kinds of those variables. The finite map ρ defines a substitution for the variables when τ appears outside of the scope

[1]Unlike other languages with intensional type analysis such as λ_i^{ML} [11] and λ_R [6], LH does not include *Typerec*—a type constructor that defines other *types* by intensional analysis.

[2]There are no type constructors analogous to polymorphic types ($\forall\alpha{:}\kappa.\sigma$) when κ is not $\star$. Including them would require either an infinite number of operators or kind polymorphism.

$$
\begin{array}{l}
\mathit{size} = \\
\quad \Lambda\alpha{:}\star \to \star.\ \mathit{typerec}\,[\emptyset,\emptyset,\emptyset][\lambda\beta{:}\star.\beta \to \mathit{int}]\langle\alpha : \star \to \star\rangle\ \mathit{of}\ \theta
\end{array}
$$

$$
\begin{array}{lll}
\multicolumn{3}{l}{\text{where } \theta =} \\
\{\ \mathit{int} & \Rightarrow & \lambda y{:}\,\mathit{int}\,.0 \\
\quad \mathit{unit} & \Rightarrow & \lambda y{:}\,\mathit{unit}\,.0 \\
\quad \times & \Rightarrow & \Lambda\beta{:}\star.\lambda x{:}(\beta \to \mathit{int}).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to \mathit{int}). \\
 & & \quad \lambda v{:}(\beta \times \gamma). \\
 & & \qquad x(\pi_1 v) + y(\pi_2 v) \\
\quad \to & \Rightarrow & \mathit{undefined} \\
\quad + & \Rightarrow & \Lambda\beta{:}\star.\lambda x{:}(\beta \to \mathit{int}).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to \mathit{int}). \\
 & & \quad \lambda v{:}(\beta + \gamma).\ \mathit{case}\ \ v\ \ \mathit{of} \\
 & & \qquad (\mathit{inj}_1\, z \Rightarrow x(z) \mid \ \mathit{inj}_2\, z \Rightarrow y(z)) \\
\quad \forall_\star & \Rightarrow & \mathit{undefined} \\
\quad \exists_\star & \Rightarrow & \Lambda\alpha{:}\star \to \star.\lambda r{:}(\forall\beta{:}\star.(\beta \to \mathit{int}) \to \alpha\beta \to \mathit{int}). \\
 & & \quad \lambda x{:}\exists\alpha. \\
 & & \qquad \mathit{let}\langle\beta, y\rangle = \mathit{unpack}\, x\ \mathit{in} \\
 & & \qquad\quad (r\ [\beta]\ (\lambda x{:}\beta.0)\ y) \\
\quad \mu_\star & \Rightarrow & \Lambda\alpha{:}\star \to \star. \\
 & & \quad \lambda x{:}(\forall\beta{:}\star.(\beta \to \mathit{int}) \to \alpha\beta \to \mathit{int}). \\
 & & \qquad \mathit{fix}\ f{:}\mu_\star\alpha \to \mathit{int}\,. \\
 & & \qquad\quad \lambda y{:}\mu_\star\alpha.\ (x\ [\mu_\star\alpha]\ f\ (\mathit{unroll}\, y)) \\
\}
\end{array}
$$

Figure 3: Example: *size*

defined by *typerec*. The type constructor τ' is an annotation that makes type checking syntax directed. We call it the *return type constructor* and will always use the metavariable τ' to refer to it. The type of a *typerec* term is $[\tau']\langle\rho(\tau) : \kappa\rangle$. This type uses the definition of a *polykinded type*, below.

Definition 2.1 *A* polykinded type*, written* $[\tau']\langle\tau\ :\ \kappa\rangle$*, where* τ' *has kind* $\star \to \star$ *and* τ *has kind* κ*, is defined by induction on* κ *by:*

$$
\begin{array}{l}
[\tau']\langle\tau : \star\rangle \overset{\text{def}}{=} \tau'\tau \\
[\tau']\langle\tau : \kappa_1 \to \kappa_2\rangle \overset{\text{def}}{=} \forall\alpha{:}\kappa_1.[\tau']\langle\alpha : \kappa_1\rangle \to [\tau']\langle\tau\alpha : \kappa_2\rangle
\end{array}
$$

A simple example of a *typerec* term is

$$
\begin{array}{l}
\mathit{typerec}\,[\{\alpha \Rightarrow \star\}, \{\alpha \Rightarrow 3\}, \{\alpha \Rightarrow \mathit{int}\}] \\
\quad [\lambda\beta{:}\star.\beta]\langle\alpha : \star\rangle\ \mathit{of}\ \{\mathit{int} \Rightarrow 0\}
\end{array}
$$

The environment for this term maps the type variable α to the number 3. This term has type $[\lambda\beta{:}\star.\beta]\langle\{\alpha \Rightarrow \mathit{int}\}(\alpha) : \star\rangle = ((\lambda\beta{:}\star.\beta)\ \mathit{int}) = \mathit{int}$.

The function *size* in Figure 3 is a more realistic example of a polytypic function defined with *typerec*. This function is defined over type constructors of kind $\star \to \star$. For example, lists are defined in this language as

$$
\mathit{List} \overset{\text{def}}{=} \lambda\beta{:}\star.\mu_\star(\lambda\alpha : \star.\ \mathit{unit} + (\beta \times \alpha))
$$

The type application $\mathit{size}[\mathit{List}]$ is a function that takes a method to compute the size of values of type β (i.e. a function of type $\beta \to \mathit{int}$), and returns a function to compute the size of the entire list of type $\mathit{List}\ \beta$.

In *size*, the return type constructor is $(\lambda\beta{:}\star.\beta \to \mathit{int})$ so the type of *size* is

$$
\begin{array}{l}
\forall\alpha{:}\star \to \star.[\lambda\beta{:}\star.\beta \to \mathit{int}]\langle\alpha : \star \to \star\rangle \\
\quad = \forall\alpha{:}\star \to \star.\forall\beta{:}\star.(\beta \to \mathit{int}) \to (\alpha\beta) \to \mathit{int}\,.
\end{array}
$$

$$
\boxed{\Delta;\Gamma[\tau'] \vdash \Delta' \mid \eta \mid \rho} \qquad \frac{}{\Delta;\Gamma[\tau'] \vdash \emptyset \mid \emptyset \mid \emptyset}
$$

$$
\frac{\begin{array}{c}\Delta;\Gamma[\tau'] \vdash \Delta' \mid \eta \mid \rho \\ \Delta;\Gamma \vdash e : [\tau']\langle\tau : \kappa\rangle \qquad \Delta \vdash \tau : \kappa \\ \beta \notin \mathit{Dom}(\Delta, \Delta')\end{array}}{\Delta;\Gamma[\tau'] \vdash \Delta'\{\beta \Rightarrow \kappa\} \mid \eta\{\beta \Rightarrow e\} \mid \rho\{\beta \Rightarrow \tau\}}
$$

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

$$
\frac{\begin{array}{c}\Delta \vdash \tau' : \star \to \star \qquad \Delta;\Gamma[\tau'] \vdash \Delta' \mid \eta \mid \rho \qquad \Delta,\Delta' \vdash \tau : \kappa \\ \Delta;\Gamma \vdash \theta(\oplus) : [\tau']\langle\oplus : \Sigma(\oplus)\rangle \qquad (\forall\oplus \in \mathit{Dom}(\Sigma))\end{array}}{\Delta;\Gamma \vdash \mathit{typerec}\,[\Delta',\eta,\rho][\tau']\langle\tau : \kappa\rangle\ \ \mathit{of}\ \ \theta : [\tau']\langle\rho(\tau) : \kappa\rangle}
$$

Figure 4: Static Semantics of LH *typerec*

We can use *size* to generate the length function for lists if we supply the constant function $(\lambda x{:}\beta.1)$ to compute the size of the list elements. In other words, $\mathit{length} = \Lambda\beta{:}\star.\mathit{size}[\mathit{List}][\beta](\lambda x{:}\beta.1)$. Likewise, if we would like a function that counts the number of values stored in a tree or the number of values in a *Maybe* (either 1 or 0), we replace the type constructor argument *List* above with $\mathit{Tree} \overset{\text{def}}{=} \lambda\beta{:}\star.\mu_\star(\lambda\alpha : \star.\beta + (\alpha \times \alpha))$ or $\mathit{Maybe} \overset{\text{def}}{=} \lambda\beta : \star.\ \mathit{unit} + \beta$.

The branches θ define interpretations for the operators. For the types *int* and *unit*, *size* returns the constant function 0 because we only wish to count values of type α. Because the $\times$ constructor must be applied to two types β and γ to produce a product type, its interpretation uses the size functions for β and γ to produce the size function for a product type $\beta \times \gamma$. The size of a product type is the sum of the sizes of the two components of the product. Likewise, the size function for a sum type determines the case of the sum and applies the appropriate size function. Like many polytypic functions, *size* is undefined for functions and polymorphic terms and will produce an error if these operators appear in its argument. For existential types, *size* unpacks the existential and then computes the size of the body, using the constant zero function as the size of the abstract type β. Finally, for recursive types, the argument x will compute the *size* function of the body of the recursive type if it is given the *size* function for the recursive type itself; this function is defined using *fix*.

The static semantics of LH (Figure 4) includes a judgment of the form $\Delta;\Gamma \vdash e : \sigma$ to indicate that a term e has type σ in type context Δ and term context Γ. Δ maps type variables to kinds and Γ maps term variables to types. Most of the rules for deriving this judgment are standard and are not described in this paper. We describe the rule for *typerec* below. In the expression $\mathit{typerec}\,[\Delta',\eta,\rho][\tau']\langle\tau : \kappa\rangle\ \mathit{of}\ \ \theta$, the context Δ' describes the kinds of type variables that may appear in the argument τ. The term environment η and type environment ρ are used to interpret those type variables. We check that Δ', η, and ρ are well-formed with the judgment form $\Delta;\Gamma[\tau'] \vdash \Delta' \mid \eta \mid \rho$. This judgment declares that η maps type variables in Δ' to appropriate terms for the return type constructor $[\tau']$, and that ρ maps those variables to type constructors with kind specified by Δ'. The first two inference rules in Figure 4 show when this judgment may be derived.

With this judgment, we can state the formation rule for higher-order *typerec* (the last rule in Figure 4). If the re-

$$\boxed{e \mapsto_h e'} \qquad \frac{}{\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle\alpha : \kappa\rangle\ \mathit{of}\ \ \theta \mapsto_h \eta(\alpha)}$$

$$\frac{}{\begin{array}{l}\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle(\lambda\alpha{:}\kappa'.\tau_1) : \kappa' \to \kappa\rangle\ \ \mathit{of}\ \ \theta \mapsto_h \\ \quad \Lambda\beta{:}\kappa'.\lambda x{:}[\tau']\langle\beta : \kappa'\rangle. \\ \qquad \mathit{typerec}\,[\Delta'\{\alpha \Rightarrow \kappa'\}, \eta\{\alpha \Rightarrow x\}, \rho\{\alpha \Rightarrow \beta\}][\tau']\langle\tau_1 : \kappa\rangle\ \mathit{of}\ \ \theta\end{array}}$$

$$\frac{\Delta' \vdash \tau_2 : \kappa'}{\begin{array}{l}\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle\tau_1\tau_2 : \kappa\rangle\ \ \mathit{of}\ \ \theta \mapsto_h \\ \quad (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle\tau_1 : \kappa' \to \kappa\rangle\ \ \mathit{of}\ \ \theta)\ [\rho(\tau_2)]\ (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle\tau_2 : \kappa'\rangle\ \mathit{of}\ \ \theta)\end{array}}$$

$$\frac{}{\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle\oplus : \Sigma(\oplus)\rangle\ \mathit{of}\ \ \theta \mapsto_h \theta(\oplus)}$$

Figure 5: Dynamic semantics of LH *typerec*

turn type constructor is well formed, the environments are well formed, the argument τ is well formed (with context extended by Δ') and all branches are described by the appropriate polykinded type (where $\Sigma(\oplus)$ is the kind of $\oplus$), then the *typerec* term is well formed.

The operational semantics for *typerec* (Figure 5) precisely describes how *typerec* interprets its argument τ. If τ is a type variable α, *typerec* looks up the interpretation of that variable in the environment η. If τ is a type function $(\lambda\alpha{:}\kappa.\tau_1)$, *typerec* steps to a polymorphic term function that, after receiving x (the interpretation of α), interprets τ_1. If τ is a type application $\tau_1\tau_2$, *typerec* steps to an application of the interpretation of τ_1 to the type τ_2 and its interpretation. Because τ_2 escapes the scope of *typerec* in the type application, we use ρ to substitute for the variables. If τ is an operator $\oplus$, *typerec* retrieves that branch from θ.

For example, Haskell's *Maybe* (or ML's *option*) type constructor is defined as $\lambda\alpha{:}\,\mathit{unit} + \alpha$. We can use *size* to define a function that returns 0 when no data is present (the first case of the sum) and 1 otherwise. The expression $\mathit{size}[\mathit{Maybe}][\mathit{unit}](\lambda x : \mathit{unit}\,.1)$ does so for arguments of type $\mathit{Maybe}\,\mathit{unit}$. We can trace the evaluation of this term as follows (Let Ψ abbreviate the context and environments $\{\alpha \Rightarrow \star\}, \{\alpha \Rightarrow (\lambda x{:}\,\mathit{unit}\,.1)\}, \{\alpha \Rightarrow \mathit{unit}\}$, let θ be the branches for *size* and let τ' be the return type constructor $(\lambda x{:}\alpha.\alpha \to \mathit{int})$):

$$\begin{array}{l}
\mathit{size}[\lambda\alpha{:}\star.\alpha + \mathit{unit}][\mathit{unit}](\lambda x : \mathit{unit}\,.1) \\
\quad \mapsto_h (\Lambda\beta{:}\star.\lambda w{:}(\alpha \to \mathit{int}).\lambda v{:}(\alpha \times \mathit{unit}). \\
\qquad \mathit{typerec}[\{\alpha \Rightarrow \star\}, \{\alpha \Rightarrow w\}, \{\alpha \Rightarrow \beta\}][\tau'] \\
\qquad\quad \langle\alpha + \mathit{unit} : \star\rangle\ \mathit{of}\ \ \theta)[\mathit{unit}](\lambda x{:}\,\mathit{unit}\,.1) \\
\quad \mapsto_h \mathit{typerec}[\Psi][\tau']\langle\alpha + \mathit{unit} : \star\rangle\ \mathit{of}\ \ \theta \\
\quad \mapsto_h (\mathit{typerec}[\Psi][\tau']\langle + : \star \to \star \to \star\rangle) \\
\qquad [\{\alpha \Rightarrow \mathit{unit}\}(\alpha)](\mathit{typerec}[\Psi][\tau']\langle\alpha : \star\rangle) \\
\qquad [\{\alpha \Rightarrow \mathit{unit}\}(\mathit{unit})](\mathit{typerec}[\Psi][\tau']\langle\mathit{unit} : \star\rangle \\
\quad \mapsto_h (\Lambda\beta{:}\star.\lambda x{:}(\beta \to \mathit{int}).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to \mathit{int}). \\
\qquad \lambda v{:}(\beta + \gamma).\ \ \mathit{case}\ \ v\ \ \mathit{of} \\
\qquad\quad (\mathit{inj}_1\, z \Rightarrow x(z)\ \mid\ \mathit{inj}_2\, z \Rightarrow y(z))) \\
\qquad [\mathit{unit}]\ \mathit{typerec}[\Psi][\tau']\langle\alpha : \star\rangle[\mathit{unit}]\ \mathit{typerec}[\Psi][\tau']\langle\mathit{unit} : \star\rangle \\
\quad \mapsto_h \lambda v{:}(\mathit{unit} + \mathit{unit}).\ \mathit{case}\, v\ \ \mathit{of} \\
\qquad (\mathit{inj}_1\, z \Rightarrow (\mathit{typerec}[\Psi][\tau']\langle\alpha : \star\rangle)(z) \\
\qquad \mid\ \mathit{inj}_2\, z \Rightarrow (\mathit{typerec}[\Psi][\tau']\langle\mathit{unit} : \star\rangle)(z))
\end{array}$$

Reduction shows that this result is equivalent to: $\lambda v{:}(\mathit{unit} + \mathit{unit}).\ \mathit{case}\, v\ \ \mathit{of}\ \ (\mathit{inj}_1\, z \Rightarrow 1\ \mid\ \mathit{inj}_2\, z \Rightarrow 0)$

3 The problem with type constructor representations

The LH language requires a type-passing semantics. The operation semantics of *typerec* examines type constructors that must be present at run-time. However, for many reasons we might want to add the facilities of higher-order *typerec* to a language with a type-erasure semantics. Crary Weirich and Morrisett [6] (CWM) defined the λ_R language that has a type-erasure semantics and operations for first-order type analysis. We can use ideas from that language to define a type-erasure language that supports higher-order analysis.

In λ_R, *typerec* analyzes terms that represent types instead of types. A special type $R\ \tau$ is the type of the representation of τ. This language also includes term constants to represent types, such as R_{int} that represents the integer type and so has type $R\ \mathit{int}$, and $R_\times$ that represents $\tau_1 \times \tau_2$ when applied to the representations of τ_1 and τ_2. $R_\times$ has type $\forall\alpha{:}\star.R\alpha \to \forall\beta{:}\star.R\beta \to R(\alpha \times \beta)$.

CWM define representations for the entire type constructor language, including higher-order type constructors, so that it is conceivable that we could extend CWM's *typerec* to the representations of higher-order type constructors. The execution of higher-order *typerec* in LH depends on the syntactic form of its type constructor argument: whether it is a variable α, a function $\lambda\alpha{:}\kappa.\tau$, an application $\tau_1\tau_2$ or a constant (such as *int* or $\to$). It would seem reasonable for a type-erasure *typerec* to determine whether the syntactic form of its argument is the representation of a variable, the representation of a function, the representation of an application or the representation of a constant.

However, there is a problem with this idea. Not all terms with representation types are syntactically equal to the representation of some type constructor. CWM represent a type variable with a term variable, a type function with a polymorphic term function, a type application with term application, and a type operator with a new representation constant. More specifically, $\mathcal{R}[\![\tau]\!]$, the representation of the type τ is defined as:

$$\begin{array}{rcl}
\mathcal{R}[\![\alpha]\!] & = & x_\alpha \\
\mathcal{R}[\![\lambda\alpha{:}\kappa.\tau]\!] & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}[R]\langle\alpha : \kappa\rangle.\mathcal{R}[\![\tau]\!] \\
\mathcal{R}[\![\tau_1\tau_2]\!] & = & (\mathcal{R}[\![\tau_1]\!])[\tau_2](\mathcal{R}[\![\tau_2]\!]) \\
\mathcal{R}[\![\oplus]\!] & = & R_\oplus
\end{array}$$

The type of a representation term is determined by the kind of the constructor it represents. If τ has kind κ, then $\mathcal{R}[\![\tau]\!]$ has the polykinded type $[R]\langle\tau : \kappa\rangle$. However, because other

$\boxed{\tau \rightsquigarrow^{wh} p}$

$$\frac{}{\alpha \rightsquigarrow^{wh} \alpha} \qquad \frac{}{\oplus \rightsquigarrow^{wh} \oplus}$$

$$\frac{\tau_1[\tau_2/\alpha] \rightsquigarrow^{wh} p}{(\lambda\alpha{:}\kappa.\tau_1)\tau_2 \rightsquigarrow^{wh} p} \qquad \frac{\tau_1 \rightsquigarrow^{wh} p}{\tau_1\tau_2 \rightsquigarrow^{wh} p\ \tau_2}$$

$\boxed{e \Rightarrow_k e'}$

$$\frac{\Delta'(\alpha) = \kappa_1 \to \ldots \to \kappa_n \to \star}{\begin{array}{l} \mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \alpha\ \tau_1 \ldots \tau_n : \star\rangle\ \mathit{of}\ \ \theta \\ \quad \Rightarrow_k \eta(\alpha)\ [\rho(\tau_1)]\ (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau_1 : \kappa_1\rangle\ \mathit{of}\ \ \theta) \ldots \\ \qquad\qquad [\rho(\tau_n)]\ (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau_n : \kappa_n\rangle\ \mathit{of}\ \ \theta) \end{array}}$$

$$\frac{\Sigma(\oplus) = \kappa_1 \to \ldots \to \kappa_n \to \star}{\begin{array}{l} \mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \oplus\ \tau_1 \ldots \tau_n : \star\rangle\ \mathit{of}\ \ \theta \\ \quad \Rightarrow_k \theta(\oplus)\ [\rho(\tau_1)]\ (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau_1 : \kappa_1\rangle\ \mathit{of}\ \ \theta) \ldots \\ \qquad\qquad [\rho(\tau_n)]\ (\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau_n : \kappa_n\rangle\ \mathit{of}\ \ \theta) \end{array}}$$

$\boxed{e \mapsto_k e'}$

$$\frac{\begin{array}{c} \tau \rightsquigarrow^{wh} p \\ \mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle p : \star\rangle\ \mathit{of}\ \ \theta \Rightarrow_k e \end{array}}{\mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau : \star\rangle\ \mathit{of}\ \ \theta \mapsto_k e}$$

$$\frac{}{\begin{array}{l} \mathit{typerec}\,[\Delta', \eta, \rho][\tau']\langle \tau : \kappa_1 \to \kappa_2\rangle\ \mathit{of}\ \ \theta \\ \quad \mapsto_k \Lambda\beta{:}\kappa_1.\ \lambda x{:}[\tau']\langle \beta : \kappa_1\rangle. \\ \qquad\quad \mathit{typerec}[\Delta'\{\gamma \Rightarrow \kappa_1\}, \eta\{\gamma \Rightarrow x\}, \rho\{\gamma \Rightarrow \beta\}] \\ \qquad\qquad [\tau']\langle \tau\gamma : \kappa_2\rangle\ \mathit{of}\ \ \theta \end{array}}$$

Figure 6: Kind-directed operational semantics

$$\begin{array}{lll}
\textit{(kinds)} & \kappa & ::= \star \mid \kappa_1 \to \kappa_2 \\
\textit{(operators)} & \oplus & ::= \mathit{int} \ \mid\ \to\ \mid \forall_\star \\
\textit{(type con's)} & \tau & ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 \mid \oplus \\
\textit{(types)} & \sigma & ::= \tau \mid \ \mathit{int}\ \mid \sigma_1 \to \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \ \mid R\tau'\tau \\
\textit{(operator rep's)} & R_\oplus & ::= \ R_{int} \mid R_\to \mid R_{\forall_\star} \\
\textit{(terms)} & e & ::= i \mid x \mid \lambda x{:}\sigma.e \mid e_1e_2 \\
 & & \quad \mid \ \ \Lambda\alpha{:}\kappa.v \ \mid e[\tau] \mid \ R_\oplus \\
 & & \quad \mid \ \ \mathit{typerec}[\tau']\ e\ \ \mathit{of}\ \ \theta \mid \ \mathit{untyrec}[\tau']\ e \\
\textit{(values)} & v & ::= i \mid \lambda x{:}\sigma.e \mid \Lambda\alpha{:}\kappa.v \\
 & & \quad \mid \ \ p \mid \ \mathit{untyrec}[\tau']\ e \\
\textit{(paths)} & p & ::= \ R_\oplus[\tau'] \mid p\ [\tau]\ e_1\ e_2
\end{array}$$

Figure 7: Syntax of LHR

terms besides $\mathcal{R}[\![\tau]\!]$ have type $[R]\langle\tau : \kappa\rangle$, it is difficult to define an operational semantics for *typerec* based on matching $\mathcal{R}[\![\tau]\!]$. Consider trying to match the representation of a type function. The type of the argument is the representation of a constructor of kind $\kappa \to \kappa'$ so it has type $\forall{:}\kappa.[R]\langle\alpha : \kappa\rangle \to [R]\langle\tau\alpha : \kappa'\rangle$. The type-erasure version of *typerec* must determine if that argument is exactly a type abstraction surrounding a term abstraction, a variable, a representation constant or an application of a representation to a type and another representation. These rules do not cover every case. For example, the term

$$\Lambda\alpha{:}\kappa.((\lambda y{:}[R]\langle\alpha : \kappa\rangle \to [R]\langle\tau\alpha : \kappa'\rangle.y)(\lambda x_\alpha{:}[R]\langle\alpha : \kappa\rangle.e))$$

has type $\forall{:}\kappa.[R]\langle\alpha : \kappa\rangle \to [R]\langle\tau\alpha : \kappa'\rangle$. Even if the operational semantics evaluates the argument before analyzing it with *typerec*, it will still not produce a syntactic λ as the subterm of the type abstraction. Because evaluation will not reduce the application under the type abstraction, this term will be stuck and evaluation of the *typerec* will not continue.

We solve this problem by reconsidering the operational semantics of *typerec*. We can redefine the operational semantics of *typerec* so that we never have to determine whether its argument is a syntactic type function. (See the relation $\mapsto_\kappa$ in Figure 6.) This new semantics first determines the kind of the argument to *typerec*. If that argument is of kind type, it cannot be a type function. Therefore, we weak-head normalize it and then use the relation $\Rightarrow_\kappa$ to examine its syntax.

If the argument to *typerec* has a function kind then we make the following observation: Because *typerec* in LH *interprets* a type constructor, it is not important whether it analyzes the type constructor τ or its eta-expansion $(\lambda\alpha : \star.\tau\ \alpha)$. Both arguments to *typerec* should produce the same result. Because something of a function kind is always equivalent to a literal type function, we know it will always step to a term function in LH. So with this semantics, an argument of function kind will always step to a term function. Though it may proceed in a different evaluation order than that of LH, this operational semantics will eventually produce the same result (see [32] for a formalization and proof of this statement.)

In a type-erasure language, we do not want to make the operational semantics depend on any type information, including its kind. However, because that kind is known at compile-time, higher-order *typerec* is definable as a "macro" in the erasure language. A *typerec* on an argument of kind $\kappa_1 \to \kappa_2$ can always be replaced by a *typerec* on argument of κ_2. As a result, the erasure language restricts analysis to arguments that represent constructors of kind $\star$.

An additional concern is one of linguistic complexity. Because the type-passing version of *typerec* examines arguments with type variables, we need to evaluate terms with free term variables (the representations of those type variables.) Extending the semantics to include the evaluation of open terms would require many new rules and the implementation of such a calculus would be complicated.

Instead, there is a simpler way to define the type-erasure calculus, based on an implementation of induction over higher-order abstract syntax [8, 29]. To avoid evaluating representations with free term variables, we change how *typerec* interprets type variables. Instead of using an environment to store the interpretations of variables, we use substitution. We add a special inverse operator (called *untyrec*) to immediately substitute the interpretation of a variable for its representation.

4 LHR: Higher-order analysis in a type-erasure language

Figure 7 shows the syntax of the LHR language. This language has a type-erasure semantics. Unlike Figure 5, no rule in the dynamic semantics of *typerec* (Figure 8) examines the syntax of a type constructor. Instead, *typerec* analyzes the term representations of type constructors formed from the representations of the operators R_{int}, $R_\to$ and $R_{\forall_\star}$. Furthermore, in this language *typerec* may only analyze the representations of constructors of kind $\star$.

Each rule for a specific operator of LHR in Figure 8 is generated from the following general rule that corresponds

$\boxed{e \mapsto_{LHR} e'}$

$$\mathit{typerec}[\tau']\ (\mathit{untyrec}[\tau']\ e)\ \mathit{of}\ \ \theta \mapsto_{LHR} e$$

$$\mathit{typerec}[\tau']\ (R_{int})\ \mathit{of}\ \ \theta \mapsto_{LHR} \theta(\mathit{int})$$

$$\begin{array}{l} \mathit{typerec}[\tau']\ (R_{\rightarrow}\ [\tau'][\tau_1]\ e'_{\tau_1}\ e_{\tau_1}\ [\tau_2]\ e'_{\tau_2}\ e_{\tau_2})\ \mathit{of}\ \ \theta \\ \quad \mapsto_{LHR} \theta(\rightarrow)\ [\tau_1]\ e'_{\tau_1}\ (\mathit{typerec}[\tau']\ e_{\tau_1}\ \mathit{of}\ \ \theta) \\ \qquad [\tau_2]\ e'_{\tau_2}\ (\mathit{typerec}[\tau']\ e_{\tau_2}\ \mathit{of}\ \ \theta) \end{array}$$

$$\begin{array}{l} \mathit{typerec}[\tau']\ (R_{\forall_\star}\ [\tau'][\tau_1]e'_{\tau_1}e_{\tau_1})\ \mathit{of}\ \ \theta \\ \qquad \mapsto_{LHR} \theta(\forall_\star)\ [\tau_1]\ e'_{\tau_1}\ (\Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta:\star\rangle.\lambda y{:}(\tau'\beta). \\ \qquad\qquad\qquad \mathit{typerec}[\tau']\ (e_{\tau_1}\ [\beta]\ x_\beta\ (\mathit{untyrec}[\tau']\ y))\ \mathit{of}\ \ \theta) \end{array}$$

$$\frac{e \mapsto_{LHR} e'}{\mathit{typerec}[\tau']\ e\ \mathit{of}\ \ \theta \mapsto_{LHR} \mathit{typerec}[\tau']\ e'\ \mathit{of}\ \ \theta}$$

Figure 8: LHR: Operational semantics of *typerec*

to $\Rightarrow_k$ evaluation of an operator of kind $\kappa_1 \rightarrow \ldots \rightarrow \kappa_n \rightarrow \star$.

$$\begin{array}{l} \mathit{typerec}[\tau']\ (R_\oplus\ [\tau']\ [\tau_1]\ e'_1\ e_1\ \ldots [\tau_n]\ e'_n\ e_n)\ \ \mathit{of}\ \ \theta \mapsto \\ \quad \theta(\oplus)\ [\tau_1]\ e'_1\ (\mathit{typerec}[\tau']\langle e_1 : \kappa_1\rangle\ \mathit{of}\ \ \theta)\ldots \\ \qquad [\tau_n]\ e'_n\ (\mathit{typerec}[\tau']\langle e_n : \kappa_n\rangle\ \mathit{of}\ \ \theta) \end{array}$$

With term representations of types and the restriction of *typerec* to the representation of types, LHR bears many similarities to λ_R. However, there is one crucial difference between this language and λ_R that allows the embedding of higher-order *typerec*. LHR includes an "inverse" operator to *typerec*, called *untyrec*. When *typerec* analyzes ($\mathit{untyrec}[\tau']e$), the embedded term e is returned. This inverse plays the role of η in higher-order *typerec* by recording the interpretation of type variables. Where we might analyze an argument with a free type variable in LH:

$$\mathit{typerec}\ [\Delta\{\beta \Rightarrow \star\}, \eta\{\beta \Rightarrow e\}, \rho\{\beta \Rightarrow \tau\}][\tau']\langle\beta : \star\rangle\ \mathit{of}\ \ \theta$$

we will translate that term to the LHR term:

$$\mathit{typerec}[\tau']\ (\mathit{untyrec}[\tau']\ e)\ \mathit{of}\ \ \theta$$

For type soundness, we must restrict what terms may be the argument to *untyrec*. Essentially, *untyrec* coerces *any* term into a representation of some type. If an arbitrary term were allowed, analysis of an *untyrec* term could result in the wrong type. The coercion is sound if we restrict the type of analysis allowed for the resulting representation. Therefore LHR parameterizes the R type with an extra argument to describe the result of type analysis allowed for that representation. When a term representation is polymorphic over this return type constructor (for example, if it is of type $\forall\beta{:}\star \rightarrow \star.R\beta\tau$) then it may be used for *any* analysis. We use the notation $\widehat{R}\langle\tau : \kappa\rangle$ as type of the representation of τ of kind κ that may be used for any analysis. It is defined (in the next section) to be $\forall\beta{:}\star \rightarrow \star.\mathcal{T}[\![[R\beta]\langle\tau : \kappa\rangle]\!]$, the type translation of the polykinded representation type. As a technicality, we trivially add R types to the source language and extend the type translation in the obvious way so that we can use this definition.

The *untyrec* term allows us to implement higher-order *typerec*. Consider analyzing the *List* type constructor in LH:

$$\mathit{typerec}\ [\Delta, \eta, \rho][\tau']\langle \mathit{List} : \star \rightarrow \star\rangle\ \mathit{of}\ \ \theta$$

$\boxed{\Delta \vdash \sigma}$

$$\frac{\Delta \vdash \tau' : \star \rightarrow \star \qquad \Delta \vdash \tau : \star}{\Delta \vdash R\ \tau'\ \tau}$$

$\boxed{\Delta \vdash \sigma = \sigma'}$

$$\frac{\Delta \vdash \tau' = \tau'' : \star \rightarrow \star \qquad \Delta \vdash \tau_1 = \tau_2 : \star}{\Delta \vdash R\ \tau'\ \tau_1 = R\ \tau''\ \tau_2}$$

$$\frac{\Delta \vdash \tau : \star \rightarrow \star}{\Delta \vdash \forall_\star\tau = \forall\alpha{:}\star.\widehat{R}\langle\alpha : \star\rangle \rightarrow \tau\alpha}$$

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

$$\frac{\Delta \vdash \oplus : \Sigma(\oplus)}{\Delta; \Gamma \vdash R_\oplus : \widehat{R}\langle\oplus : \Sigma(\oplus)\rangle}$$

$$\frac{\begin{array}{c} \Delta \vdash \tau : \star \\ \Delta \vdash \tau' : \star \rightarrow \star \\ \Delta; \Gamma \vdash \theta(\oplus) : \mathcal{T}[\![[\tau']\langle\oplus : \Sigma(\oplus)\rangle]\!] \qquad (\forall\oplus \in \mathit{Dom}(\Sigma)) \\ \Delta; \Gamma \vdash e : R\ \tau'\tau \end{array}}{\Delta; \Gamma \vdash \mathit{typerec}[\tau']\ e\ \ \mathit{of}\ \ \theta : \tau'\tau}$$

$$\frac{\begin{array}{c} \Delta \vdash \tau : \star \\ \Delta \vdash \tau' : \star \rightarrow \star \\ \Delta; \Gamma \vdash e : \tau'\tau \end{array}}{\Delta; \Gamma \vdash \mathit{untyrec}[\tau']\ e\ \ : R\ \tau'\tau}$$

Figure 9: LHR: Static Semantics of *typerec*

In LHR, the representations of type constructors of higher kinds are term functions. For example, if e_{List} is the representation of *List* then it is a function from the representation of some type α to the representation of the type *List* α. Therefore, in LHR, we can analyze the list constructor with a term that abstracts the interpretation of α and then analyzes the result of applying e_{List} to *untyrec* surrounding that interpretation.

$$\begin{array}{l} \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star\rangle.\lambda y{:}(\tau'\alpha). \\ \quad \mathit{typerec}[\tau']\ (e_{List}\ [\alpha]\ x_\alpha\ (\mathit{untyrec}[\tau']\ y))\ \mathit{of}\ \ \theta. \end{array}$$

LHR does not include a higher-order version of *typerec* because it may encode such terms. If e_τ is the representation of the type τ of kind κ, the general encoding of the analysis of e_τ, notated $\mathit{typerec}\ [\tau']\langle e_\tau : \kappa\rangle\ \mathit{of}\ \ \theta$ is in Figure 11. This operation is defined in conjunction with its inverse, a higher-order version of *untyrec*. Both operations are defined by induction on κ, the kind of the represented type constructor.

Figure 9 shows the static semantics for the representation terms, *typerec* and *untyrec*. If $\oplus$ is an arbitrary type constructor constant, such as *int*, $\rightarrow$, $\forall_\star$ in LHR, $R_\oplus$ is its term representation. If $\oplus$ is of kind $\Sigma(\oplus)$, then the type of $R_\oplus$ is $\widehat{R}\langle\oplus : \Sigma(\oplus)\rangle$. The argument to *typerec* is the representation of a type. The types of the branches of *typerec* are the translations of the types of the LH branches. The result of *untyrec* is also a term representation.

So that we may prove the correctness of this translation, we must change the type equivalence rule for the operator $\forall_\star$ to witness the type translation. However, it is also possible to add the operator $\hat{\forall}_\star$ to this calculus, such that

$$
\begin{array}{rcl}
\mathit{typerec}[\tau'] \langle\!\langle e_\tau : \star \rangle\!\rangle \mathit{of}\ \theta & \stackrel{\text{def}}{=} & \mathit{typerec}[\tau']\ e_\tau\ \mathit{of}\ \ \theta \\
\mathit{typerec}[\tau'] \langle\!\langle e_\tau : \kappa_1 \to \kappa_2 \rangle\!\rangle \mathit{of}\ \theta & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa_1.\lambda x{:}\widehat{R}\langle \alpha : \kappa_1\rangle.\lambda y{:}\mathcal{T}\llbracket [\tau'] \langle \alpha : \kappa_1 \rangle \rrbracket.N \\
 & & \text{where} \\
 & & N = \mathit{typerec}[\tau'] \langle\!\langle (e_\tau[\alpha]\ x\ M) : \kappa_2 \rangle\!\rangle \mathit{of}\ \theta \\
 & & M = \mathit{untyrec}[\tau'] \langle\!\langle y : \kappa_1 \rangle\!\rangle \mathit{of}\ \theta \\
 & & \\
\mathit{untyrec}[\tau'] \langle\!\langle e_\tau : \star \rangle\!\rangle \mathit{of}\ \theta & \stackrel{\text{def}}{=} & \mathit{untyrec}[\tau']\ e_\tau \\
\mathit{untyrec}[\tau'] \langle\!\langle e_\tau : \kappa_1 \to \kappa_2 \rangle\!\rangle \mathit{of}\ \theta & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa_1.\lambda x{:}\widehat{R}\langle \alpha : \kappa_1\rangle.\lambda y{:}\mathcal{T}\llbracket [\tau'] \langle \alpha : \kappa_1 \rangle \rrbracket.N \\
 & & \text{where} \\
 & & N = \mathit{untyrec}[\tau'] \langle\!\langle (e_\tau[\alpha]\ x\ M) : \kappa_2 \rangle\!\rangle \mathit{of}\ \theta \\
 & & M = \mathit{typerec}[\tau'] \langle\!\langle y : \kappa_1 \rangle\!\rangle \mathit{of}\ \theta
\end{array}
$$

Figure 11: Higher-order *typerec*

$$
\begin{array}{l}
\mathit{size} = \Lambda\alpha{:}\star.\lambda x_\alpha{:}R(\lambda\beta{:}\star.\beta \to \mathit{int})\,\alpha. \\
\quad \mathit{typerec}[\lambda\beta{:}\star.\beta \to \mathit{int}]\ x_\alpha\ \mathit{of}\ \ \theta \\
\\
\text{where } \theta = \\
\{\ \begin{array}[t]{ll}
\mathit{int} & \Rightarrow \lambda y{:}\,\mathit{int}.0 \\
\mathit{unit} & \Rightarrow \lambda y{:}\,\mathit{unit}.0 \\
\times & \Rightarrow \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta : \star\rangle.\lambda x{:}(\beta \to \mathit{int}). \\
 & \quad \Lambda\gamma{:}\star.\lambda x_\gamma{:}\widehat{R}\langle\gamma : \star\rangle.\lambda y{:}(\gamma \to \mathit{int}). \\
 & \quad\quad \lambda v{:}(\beta \times \gamma). \\
 & \quad\quad\quad x(\pi_1 v) + y(\pi_2 v) \\
\to & \Rightarrow \mathit{undefined} \\
+ & \Rightarrow \Lambda\beta{:}\star.\lambda x_\beta{:}\widehat{R}\langle\beta : \star\rangle.\lambda x{:}(\beta \to \mathit{int}). \\
 & \quad \Lambda\gamma{:}\star.\lambda x_\beta{:}\widehat{R}\langle\gamma : \star\rangle.\lambda y{:}(\gamma \to \mathit{int}). \\
 & \quad\quad \lambda v{:}(\beta + \gamma).\ \ \mathit{case}\ \ v\ \ \mathit{of} \\
 & \quad\quad\quad (\mathit{inj}_1\, z \Rightarrow x(z)\ |\ \ \mathit{inj}_2\, z \Rightarrow y(z)) \\
\forall_\star & \Rightarrow \mathit{undefined} \\
\exists_\star & \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star \to \star\rangle. \\
 & \quad \lambda r{:}(\forall\beta{:}\star.\widehat{R}\langle\beta : \star\rangle \to (\beta \to \mathit{int}) \to \alpha\beta \to \mathit{int}). \\
 & \quad\quad \lambda x{:}(\exists\beta : \star.\widehat{R}\langle\beta : \star\rangle \times (\alpha\beta)). \\
 & \quad\quad\quad \mathit{let}\langle\beta, \langle x_\beta, y\rangle\rangle = \mathit{unpack}\ x\ \mathit{in} \\
 & \quad\quad\quad\quad (r\ [\beta]\ x_\beta\ (\lambda x{:}\beta.0)\ y) \\
\mu_\star & \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star \to \star\rangle. \\
 & \quad \lambda x{:}(\forall\beta{:}\star.\widehat{R}\langle\beta : \star\rangle \to (\beta \to \mathit{int}) \to \alpha\beta \to \mathit{int}). \\
 & \quad\quad \mathit{fix}\ f{:}(\mu_\star\alpha \to \mathit{int}). \\
 & \quad\quad\quad \lambda y{:}\mu_\star\alpha.\ (x\ [\mu_\star\alpha]\ \widehat{\mathcal{R}}\llbracket\mu_\star\alpha\rrbracket\ f\ \ (\mathit{unroll}\, y))
\end{array} \\
\quad \}
\end{array}
$$

Figure 10: Example: Erasure version of *size*

$\hat{\forall}_\star\tau = \forall\alpha{:}\star.\tau\alpha$. This operator produces the type of parametric functions that cannot analyze their type arguments. Because all types are analyzable in LH, this operator is not a part of the source language.

The static semantics agrees with the dynamic semantics of LHR.

Theorem 4.1 (Type Safety) *If $\emptyset \vdash e : \sigma$ then e either evaluates to a value or diverges.*

Proof

(Sketch) Proof follows from the usual progress and preservation theorems [33].

$$
\begin{array}{rcl}
\mathcal{T}\llbracket\tau\rrbracket & = & \tau \\
\mathcal{T}\llbracket \mathit{int} \rrbracket & = & \mathit{int} \\
\mathcal{T}\llbracket\sigma_1 \to \sigma_2\rrbracket & = & \mathcal{T}\llbracket\sigma_1\rrbracket \to \mathcal{T}\llbracket\sigma_2\rrbracket \\
\mathcal{T}\llbracket\forall\alpha{:}\kappa.\sigma\rrbracket & = & \forall\alpha{:}\kappa.\widehat{R}\langle\alpha : \kappa\rangle \to \mathcal{T}\llbracket\sigma\rrbracket \\
\mathcal{T}\llbracket R\tau\tau'\rrbracket & = & R\ \mathcal{T}\llbracket\tau\rrbracket\ \mathcal{T}\llbracket\tau'\rrbracket \\
 & & \\
\mathcal{E}\llbracket i\rrbracket & = & i \\
\mathcal{E}\llbracket\lambda x{:}\sigma.e\rrbracket & = & \lambda x{:}\mathcal{T}\llbracket\sigma\rrbracket.\mathcal{E}\llbracket e\rrbracket \\
\mathcal{E}\llbracket e_1 e_2\rrbracket & = & \mathcal{E}\llbracket e_1\rrbracket\mathcal{E}\llbracket e_2\rrbracket \\
\mathcal{E}\llbracket\Lambda\alpha{:}\kappa.e\rrbracket & = & \Lambda\alpha{:}\kappa.\lambda x{:}\widehat{R}\langle\alpha : \kappa\rangle.\mathcal{E}\llbracket e\rrbracket \\
\mathcal{E}\llbracket e[\tau]\rrbracket & = & \mathcal{E}\llbracket e\rrbracket\ [\tau]\ \widehat{\mathcal{R}}\llbracket\tau\rrbracket \\
\mathcal{E}\llbracket\ \ \mathit{typerec}[\Delta, \rho, \eta][\tau']\ \ \rrbracket & = & \mathit{typerec}[\tau'] \langle\!\langle \mathcal{R}\llbracket\tau, \varepsilon\rrbracket : \kappa \rangle\!\rangle \mathit{of}\ \ \mathcal{E}\llbracket\theta\rrbracket \\
\langle\tau : \kappa\rangle\ \mathit{of}\ \ \theta \qquad\qquad & & \\
 & & \mathit{where} \\
 & & \varepsilon = (\emptyset, \tau', (\Delta, \rho, \mathcal{E}\llbracket\eta\rrbracket, \mathcal{E}\llbracket\theta\rrbracket))
\end{array}
$$

Figure 13: Translation of LH to LHR

We end this section with an example in Figure 10: *size* written in the type-erasure language. This function analyzes x_α, the representation of the type α. Even though α must be a type, we can still use this *size* to define length for lists below, where e_{List} is the representation of *List*.

$$
\begin{array}{l}
\mathit{length} = \Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star\rangle. \\
\quad \mathit{size}[\mathit{List}\ \alpha](e_{\mathit{List}}\ [\alpha]\ x_\alpha\ (\mathit{untyrec}[\lambda\beta{:}\star.\beta \to \mathit{int}](\lambda x{:}\alpha.1)))
\end{array}
$$

There are two more differences between this version and the LH version of *size*. Whenever a type is abstracted its representation is also abstracted (for example, in the branches for $\times$ and $+$). Whenever a type is applied, its representation is also applied. (In the μ branch, the application of x to the type $[\mu_\star\alpha]$ is followed by the representation of $\mu_\star\alpha$, the term $\widehat{\mathcal{R}}\llbracket\mu_\star\alpha\rrbracket$ defined in the next section.)

5 Translating LH to LHR

The translation between LH and LHR is based on phase splitting. This process separates the static and dynamic roles of types by producing type representations in the target language for each type in the source language. The translation for types $\mathcal{T}\llbracket\sigma\rrbracket$ and terms $\mathcal{E}\llbracket e\rrbracket$ appears in Figure 13. Kinds and type constructors are unchanged. We use a number of auxiliary definitions in this translation, listed in

Type translation	$\mathcal{T}[\![\sigma]\!]$	(Figure 13)
Term translation	$\mathcal{E}[\![e]\!]$	(Figure 13)
Open representation	$\mathcal{R}[\![\tau,\varepsilon]\!]$	(Figure 14)
Parameter of open rep	Ψ	$\stackrel{\text{def}}{=} (\Delta,\rho,\eta,\theta) \mid \bullet$
Closed representation	$\widehat{\mathcal{R}}[\![\tau]\!]$	$\stackrel{\text{def}}{=} \Lambda\alpha{:}\star\to\star.\mathcal{R}[\![\tau,(\emptyset,\alpha,\bullet)]\!]$
Type of open representation	$\mathcal{T}[\![[R\tau']\langle\tau:\kappa\rangle]\!]$	Translation of polykinded type
Type of closed representation	$\widehat{R}\langle\tau:\kappa\rangle$	$\stackrel{\text{def}}{=} \forall\alpha{:}\star\to\star.\mathcal{T}[\![[R\alpha]\langle\tau:\kappa\rangle]\!]$
Higher-order *typerec*	$\mathit{typerec}[\tau']\langle e:\kappa\rangle\ \mathit{of}\ \theta$	(Section 4)
Higher-order *untyrec*	$\mathit{untyrec}[\tau']\langle e:\kappa\rangle\ \mathit{of}\ \theta$	(Section 4)
Polykinded type	$[\tau']\langle\tau:\kappa\rangle$	(Section 2)

Figure 12: Notation used in the translation

Figure 12. Most importantly, this translation replaces the argument to *typerec* with its term representation $\mathcal{R}[\![\tau,\varepsilon]\!]$ (ε contains the components of the enclosing *typerec*). To create this representation, the translation must ensure that for every type variable in scope, its term representation is also in scope. Therefore every type abstraction is immediately followed by an abstraction of its representation. Likewise, if τ is a type argument to a polymorphic term, it is followed by its representation, $\widehat{\mathcal{R}}[\![\tau]\!]$. Consequently, the type translation for polymorphic types includes the type of this representation $\widehat{R}\langle\tau:\kappa\rangle$, where κ is the kind of τ.

5.1 Representing the constructor language

There are two sorts of term representations in LHR. The first sort, called *open representations* and notated $\mathcal{R}[\![\tau_1,\varepsilon]\!]$, represent types that are arguments to *typerec*. These representations are called open because they may contain the term representation of LHR's free type variables. They are of type $\mathcal{T}[\![[R\tau']\langle\tau:\kappa\rangle]\!]$, where τ' is the return type constructor of the enclosing *typerec*. Such representations may be used *only* in an analysis that produces a result of type $\mathcal{T}[\![[\tau']\langle\tau:\kappa\rangle]\!]$. Inside these representations there may be *untyrec* terms holding the interpretations of the free variables and those results must agree with τ'.

The second sort are the *closed representations*, $\widehat{\mathcal{R}}[\![\tau]\!]$, of type $\widehat{R}\langle\tau:\kappa\rangle = \forall\beta{:}\star\to\star.\mathcal{T}[\![[R\beta]\langle\tau:\kappa\rangle]\!]$. These representations are not arguments to *typerec*—they are polymorphic with respect to the return type constructor of any possible analysis. That polymorphism must be instantiated to the appropriate type constructor before these representations may be analyzed. Because *untyrec* terms depend on this instantiation, closed representations cannot have any *untyrec* expressions as subterms.

Open and closed representations are both defined by $\mathcal{R}[\![\tau,\varepsilon]\!]$ in Figure 14. The parameter ε describes the context of this representation (Δ'), the return type constructor (τ') and (possibly) the components of a surrounding *typerec* (Ψ). In an open representation $\Psi = (\Delta,\rho,\eta,\theta)$, holding the context, environments and branches of an enclosing *typerec*. In that event, the translation constructs an appropriate higher-order *untyrec* for variables in Δ. Otherwise, for a closed representation $\Psi = \bullet$.

The tricky part of this translation is the case for variables. If the variable is in Δ and Ψ is not $\bullet$ then the variable is bound by an enclosing *typerec*, and there is a binding for it within η. This result should be wrapped by an *untyrec*. Otherwise, if the variable is in Δ', then this variable is bound by some type-level λ, and there will be a closed representative y_α already specialized to τ'. Otherwise, this variable is bound by some term-level Λ or is in a closed representation and bound by a type-level λ, and there is some closed representation x_α for it. However, x_α is polymorphic over the return type constructor, so we need to instantiate it with τ'.

The representations of type-level abstractions are polymorphic functions that abstract both the closed representations x_α and the open representations y_α. Likewise, the representations of type-level applications provide both the closed and open representations of the type argument τ_2. We use ρ to substitute for any type variables in τ_2 when we form the closed representation.

For example, $\widehat{\mathcal{R}}[\![\lambda\alpha{:}\star.\alpha\to \mathit{int}]\!]$ expands to

$$\begin{aligned}&\Lambda\beta{:}\star\to\star.\Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\star\rangle.\lambda y_\alpha{:}\mathcal{T}[\![[R\beta]\langle\alpha:\star\rangle]\!].\\ &\quad R_\to[\beta]\ [\alpha]\ x_\alpha\ y_\alpha\ [\mathit{int}]\ R_{\mathit{int}}\ (R_{\mathit{int}}[\beta])\end{aligned}$$

Here, we instantiate $R_\to$ with the return constructor β, the first component of the arrow type α, along with its closed representation x_α and its open representation y_α, and the second component of the product type *int*, along with its closed representation R_{int} and its open representation $R_{\mathit{int}}[\beta]$.

Why must $R_\to$ be applied to both the open and closed representations of its subcomponents? The branch for $\to$ in *typerec* expects both the closed representation and the iteration over the open representation for each component. Recall the dynamic semantics for this branch:

$$\begin{aligned}&\mathit{typerec}[\tau']\ (R_\to\ [\tau'][\tau_1]\ e'_{\tau_1}\ e_{\tau_1}\ [\tau_2]\ e'_{\tau_2}\ e_{\tau_2})\ \mathit{of}\ \ \theta\\ &\quad\mapsto_{LHR}\theta(\to)\ [\tau_1]\ e'_{\tau_1}\ (\mathit{typerec}[\tau']\ e_{\tau_1}\ \mathit{of}\ \ \theta)\\ &\qquad\qquad\qquad [\tau_2]\ e'_{\tau_2}\ (\mathit{typerec}[\tau']\ e_{\tau_2}\ \mathit{of}\ \ \theta)\end{aligned}$$

We cannot generate the closed representations from the open representations, yet we must produce them as the $\theta(\to)$ branch may use them as the arguments to other polytypic functions.

6 Implementation

In this section we describe an implementation of a simplified version of LHR in Haskell. [3] The interface to this implementation is the following:

[3]This implementation requires the extensions of first-class polymorphism and existential types [18] supported by the implementations GHC and Hugs.

$$
\begin{array}{rcll}
\mathcal{R}[\![\oplus,\varepsilon]\!] & = & R_{\oplus}[\tau'] & \\
\mathcal{R}[\![\alpha,\varepsilon]\!] & = & \begin{cases} \mathit{untyrec}[\tau'](\!|\eta(\alpha):\Delta(\alpha)|\!)\ \mathit{of}\ \theta & \text{if } \Psi \text{ is } (\Delta,\rho,\eta,\theta) \text{ and } \alpha \in \mathit{Dom}\,\Delta \\ y_\alpha & \text{if } \alpha \in \mathit{Dom}\,\Delta' \\ x_\alpha[\tau'] & \text{otherwise} \end{cases} & \\
\mathcal{R}[\![\lambda\alpha{:}\kappa.\tau_1,\varepsilon]\!] & = & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha:\kappa\rangle.\lambda y_\alpha{:}\mathcal{T}[\![[R\tau'](\!|\alpha:\kappa|\!)]\!].\mathcal{R}[\![\tau_1,(\Delta'\{\alpha\Rightarrow\kappa\},\tau',\Psi)]\!] & \\
\mathcal{R}[\![\tau_1\tau_2,\varepsilon]\!] & = & \mathcal{R}[\![\tau_1,\varepsilon]\!]\ [\rho(\tau_2)]\ \widehat{\mathcal{R}}[\![\rho(\tau_2)]\!]\ \mathcal{R}[\![\tau_2,\varepsilon]\!] & \\
& & & \\
\text{where } \varepsilon & = & (\Delta',\tau',\Psi) &
\end{array}
$$

Figure 14: Representation of constructor language

```
type R c a

rint   :: R c Int
runit  :: R c ()
rtimes :: R c a -> R c b -> R c (a, b)
rname  :: (String, [DataCon (R c) b]) -> R c b
rex    :: (forall b. R c b -> R c (a b))
            -> R c (Ex a)

typerec :: Theta c -> R c a -> c a
untyrec :: c a -> R c a

data Theta c = Theta {
  int   :: c Int,
  unit  :: c (),
  times :: forall a b. R c a -> R c b -> c (a, b),
  name  :: forall b. (String, [DC (R c) b])
            -> c b,
  ex    :: forall a. (forall b. R c b -> R c (a b))
            -> c (Ex a)
}
```

This implementation includes definitions of the R type constructor, constants for the representations of type operators $R_{\oplus}$, the *untyrec* operator, and the type analysis operator *typerec*. The datatype `Theta` is a record that describes the types of the branches to `typerec`.

The `name` branch in `Theta` is for the analysis of Haskell data types and newtypes. These type forms represent recursive types such as lists and trees. There is a list of `DC`s in the argument to the `name` branch that corresponds to the constructors of the datatype.

```
data DC c a =
  forall b. DC String (c b) (b -> a) (a -> Maybe b)
```

For each data constructor, This datatype contains the name of that constructor, the result of `typerec` for the argument of that constructor (for uniformity we uncurry data constructors), the constructor itself, and a "matching" function to determine if an element of type `a` is the specified constructor.

For example, we represent the list type constructor by a term function. This function uses `rname` to create a representation of `[a]` given the information about the named type: the string `"List"` and the representations of the data constructors nil and cons. The string can be used to augment a generic function with a special case for a particular named type.

```
rlist :: R c a -> R c [a]
rlist ra = rname ("List", [rnil, rcons ra])
```

```
rnil :: DC (R c) [a]
rnil = DC "[]" runit (\x -> [])
                     (\x -> case x of
                        [] -> Just ()
                        (_) -> Nothing)
rcons :: R c a -> DC (R c) [a]
rcons ra = DC ":" (rtimes ra (rlist ra))
                  (\(t1,t2) -> t1 : t2)
                  (\x -> case x of
                       (t1:t2) -> Just(t1,t2)
                       (_) -> Nothing)
```

The last branch of `Theta` is for existential types. We use the following datatype to represent an existential type that includes the closed representation of the hidden type (i.e. `∃a.(∀c. R c a × f a)`).

```
data Ex f = forall a. Ex (forall c. (R c a, f a))
```

We could also omit the closed representation from the existential type constructor but the polytypic operations that we could instantiate with this constructor are limited because we do not have access to the representation of the hidden type variable.

The difference between this interface and LHR is that here the branches for *typerec* do not provide the closed representation of the subcomponents of the types or the result of *typerec* for that subcomponent. Otherwise, the type of the `times` branch would be:

```
times :: forall a b.
             (forall c. R c a) -> R c a -> c a
          -> (forall c. R c b) -> R c b -> c b
          -> c (a, b)
```

This omission means that type representations also do not carry closed representations. Extending this implementation to include those representations is tedious but not difficult. Furthermore, open representations would allow our polytypic operations to be defined in terms of other polytypic operations.

However, even without closed representations, we have enough information to implement the size example. To pass the return type constructor ($\lambda\alpha{:}\star.\alpha \to \mathit{int}$) as an argument to the `R` type constructor requires that we first give it a name with a newtype. (Haskell does not allow type-level lambdas).

```
newtype Size a = S (a -> Int)
unS (S a) = a

size :: R Size a -> a -> Int
size ra = unS . (typerec theta_size ra)
```

The branches for size are very similar to the ones in Figure 10, except for the coercions into the newtype Size. For example, in the int branch, we use S to coerce the constant zero function to be of type Size Int.

```
theta_size :: Theta Size
theta_size = Theta {
  int   = S (\x -> 0),
  unit  = S (\x -> 0),
  times = \xa xb -> S $ \v ->
           size xa (fst v) + size xb (snd v),
  name  =  (string, cons) ->
              S $ \v ->
              let loop (DC _ xa inn out: rest) =
                     case (out v) of
                        Just y -> size xa y
                        Nothing -> loop rest
                  loop [] = error "impossible"
              in loop cons,
  ex    = \xa -> S $ \(Ex w) ->
             let (rep,z) = w in
             size (xa (untyrec (S $ \x -> 0))) z
  }
```

As before, we can use size to implement length for lists by using $(\lambda x{:}\alpha.1)$ as the size function for α.

```
length :: [a] -> Int
length = size (rlist (untyrec (S $ \x -> 1)))
```

We can apply length to Haskell lists. For example, length [1,2,3] = 3.

We can also use this facility to implement first-order polytypic operations (such as those usually implemented by type classes). For example, instead of defining the Show type class, we can implement rshow:

```
newtype RepShow a = RS (a -> String)
unRS (RS a) = a
rshow :: R RepShow a -> a -> String
rshow ra = unRS . (typerec theta_show ra)
theta_show :: Theta RepShow
theta_show = Theta {
  int   = RS showInt,
  unit  = RS (const "()"),
  times = \xa xb -> RS $ \v ->
           "(" ++ rshow xa (fst v) ++ ","
               ++ rshow xb (snd v) ++ ")",
  name  = \(string, cons) ->
             RS $ \v ->
             let loop (DC str xa inn out : rest) =
                   case (out v) of
                     Just s  ->
                       let s' = rshow xa s in
                       if s' == "()" then  str
                       else str ++ " " ++ s'
                     Nothing -> loop rest
                 loop [] = error "impossible"
                 in loop cons,
  ex    = \xa -> RS $ \ (Ex w) ->
             let (rep,z) = w in
             rshow (xa rep) z
}

rshow (rlist rint) [1, 2, 3]
 = ": (1,: (2,: (3,[])))"
```

The result of rshow is different from how we might want to display lists because rshow does not use infix notation or precedence rules. Below, we describe how to modify rshow to use infix. (It is also possible to account for precedence). To show cons with infix, we change the case for data constructors above so that it checks the string to see if it is cons (:). If so, we use the polytypic infixshow to show the argument to cons. We are able to call infixshow because it returns the same type of result as rshow and so we can call it with the open representation. For most flexibility in calling other polytypic functions, we need the closed representations.

```
  case (out v) of
      Just s  ->
          if str == ":" then infixshow xa s
          else let s' = rshow xa s in  ....
```

The infixshow function behaves just like rshow except that in the case of a pair it shows the first component, then ":" and then the second component.

```
infixshow :: R RepShow a -> a -> String
infixshow = unRS . (typerec (theta_show {
  times = \xa xb -> RS $ \v ->
      "(" ++ rshow xa (fst v) ++ "):("
          ++ rshow xb (snd v) ++ ")"}))

rshow (rlist rint) [1, 2, 3]
  = "(1):((2):((3):([])))"
```

Unlike type classes, rshow extends to existential types. An extension to type classes that supports existential types would still be problematic because it would only work for existentials that contain the right dictionaries. Because this version requires a general representation of the type instead of a specific dictionary, we can use it for existentials.

For example, we can represent the type $\exists\alpha.\, \mathit{int} \times \alpha$ with:

```
type Hidden = Ex ( (,) Int)
rhidden :: R c Hidden
rhidden = rex (rtimes rint)

hidden_int :: Hidden
hidden_int = Ex (rint, (3, 4))
```

The branch for existentials prints out the entire term, including those parts with the abstract type. For example, rshow rhidden hidden_int = "(3,4)". However, the branch for existentials can also hide components of abstract type by providing a constant function:

```
ex = \xa -> RS $ \ (Ex w) ->
      let (rep,z) = w in
        rshow (xa (untyrec (RS $ const "XXX" ))) z
```

With the above branch, any values of the abstract type appear as "XXX". In other words, rshow rhidden hidden_int = "(3,XXX)".

We implement type representations in Haskell in a manner similar to representing Church numerals—each type representation is implemented as its elimination form. Because of that, we define the R type to be a function from the record of typerec branches to the return type.

```
newtype R c b = R (Theta c -> c b)
```

The implementation of `typerec` applies its representation argument to the branches to get the result. The definition of `untyrec` takes those branches, ignores them, and returns its argument `x`.

```
typerec :: Theta c -> R c a -> c a
typerec theta (R rep) = rep theta
untyrec :: c a -> R c a
untyrec x = R (\theta -> x)
```

The type representations each select the corresponding component from `theta`. (For each record label, Haskell defines a function with the same name that projects that label from a record.) For example, in the definition of `rint`, `int` is a function that retrieves the `int` component of `theta`. Therefore, it is of type `Theta c -> c Int`, and the `R` data constructor coerces it to be of type `R c Int`.

```
rint   :: R c Int
rint    = R int
runit  :: R c ()
runit   = R unit
```

The `times` branch of `theta` needs the representations of the two subcomponents `t1` and `t2`. The `name` branch needs the name of the type and the representations of the data constructors. Furthermore, the existential branch just needs the representation of its subcomponent.

```
rtimes :: R c a -> R c b -> R c (a, b)
rtimes t1 t2 = R (\theta -> times theta t1 t2)
rname  :: (String, [DC (R c) b]) -> R c b
rname   = \(str,cons) -> R (\x -> name x (str,cons))
rex    :: (forall b. R c b -> R c (a b))
              -> R c (Ex a)
rex t1 = R (\theta -> ex x t1)
```

7 Extensions

LH is only a subset of the language described by Weirich [31]. The LH language is lacking two features that complicate (but do not prohibit) the translation to the type-erasure language. The first is that the full language (following Hinze [12]) generalizes polykinded types to a relation of n arguments for more expressiveness. For example, the polytypic definition of map requires two arguments and the definition of zip requires three. A type-erasure version must have multiple representations and multiple *typerec*s, one for each n. However, all of these representations and *typerec*s have the same erasure, so a direct implementation (instead of the Haskell library implementation) could use the same terms at runtime.

A second difference is that the full language includes *kind polymorphism* and extends *typerec* to constructors with polymorphic kind. There are two reasons for this extension. First, a polytypic function in LH (such as *size*) must specify and therefore restrict the kind of its type argument. This restriction is artificial in LH because *typerec* may iterate over type constructors with any kind. However, the lack of kind polymorphism does not restrict LHR, as *typerec* in LHR is not kind-polymorphic. We do not need to make a polytypic function kind-polymorphic because we can apply such a function to the representations of higher-kinded constructors by first using *untyrec*.

The second reason for kind polymorphism is that polymorphic types (universal and existential) bind type variables with many kinds. Kind polymorphism allows *typerec* to handle all such types with one branch. We believe that it is possible, though complicated, to add kind polymorphism to LHR. The complexity arises in the definition of $\mathit{typerec}[\tau']\langle e : \kappa\rangle \mathit{of}\ \theta$ and $\mathit{untyrec}[\tau']\langle e : \kappa\rangle \mathit{of}\ \theta$ when κ is an abstract kind χ. The translation to LHR must provide this information. Therefore, all kind abstractions must also abstract a term that knows how to implement *typerec* for that kind of argument.

8 Summary and related work

This paper develops a type-erasure language supporting higher-order intensional type analysis. While type-erasure versions of several other type analyzing languages have been previously developed [6, 22], several aspects of the source language made this a not-so-straightforward task.

The largest difficulty was to develop a kind-directed operational semantics for *typerec* so that we did not need to rely on the syntactic properties of the representations of higher kinds. This operational semantics is similar to Stone and Harper's language with *singleton kinds* [25], which was inspired by Coquand's approach to $\beta\eta$-equivalence for a type theory with Π types and one universe [4]. Because equivalence of constructors in Stone and Harper's language strongly depends on the kind at which they are compared, their procedure drives the kind of the compared terms to the base form before weak-head normalizing and comparing structurally.

A second issue with creating the type-erasure language was that we did not want to define a version of evaluation for terms with free variables. Instead, we chose to directly replace those variables with a place holder for the result of their interpretation. This place holder draws inspiration from the calculus of Trifonov et al. [26] who themselves refer to Fegaras and Sheard [8]. Fegaras and Sheard designed their calculus to extend catamorphisms to datatypes with parametric function spaces, employing a place holder as the trivial inverse of the iterator. Trifonov et al. adapted this idea in a type-level *Typerec* for recursive types. Like the parameterized return constructor of the R-type in this calculus, they parameterize the return kind of a *Typerec* iteration. Such an extension to LHR would allow a higher-order type-level *Typerec*. Washburn and Weirich [29] examine the general technique of using a place holder to implement induction over higher-order abstract syntax. In particular, they are able to show a close connection between using this technique in F_ω and the modal calculus of Schürmann, Despeyroux and Pfenning [23].

The result of this paper, however, is a fairly simple type-erasure language that supports higher-order type analysis. Such a language is an important step in the implementation of a system that allows *run-time* polytypic programming. The calculus that we have defined is very simple to implement: we give a prototype implementation in only a few lines. Closely related work to this paper is a new proposal for Dependency-Style Generic Haskell [15] that addresses the problem in Generic Haskell of defining polytypic operations that depend on one another. Because closed representations to the branches of polytypic operations are already provided, that capability already exists in LHR to some extent. Furthermore, by not allowing type interpretation at run-time (or any sort of general run-time type information), Generic Haskell cannot allow types to be defined in separate modules from generic operations or analyze first-class abstract types.

References

[1] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991.

[2] Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995.

[3] James Cheney and Ralf Hinze. Poor man's dynamics and generics. In Manuel M. Chakravarty, editor, *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*. ACM Press, 2002.

[4] Thierry Coquand. An algorithm for testing conversion in type theory. In Gérard Huet and Gordon Plotkin, editors, *Logical Frameworks*, pages 255–277. Cambridge University Press, 1991.

[5] Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, September 1999.

[6] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intensional polymorphism in type erasure semantics. *Journal of Functional Programming*, 12(6):567–600, November 2002.

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

[8] Leonidas Fegaras and Tim Sheard. Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). In *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pages 284–294. ACM Press, New York, 1996.

[9] 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.

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

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

[12] Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, pages 2–27, Ponte de Lima, Portugal, July 2000.

[13] Patrick Jansson and Johan Jeuring. PolyP – a polytypic programming language extension. In *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997.

[14] C. Barry Jay, Gianna Bellè, and Eugenio Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998.

[15] Andres Löh, Dave Clarke, and Johan Juering. Dependency-style Generic Haskell. In *ACM SIGPLAN International Conference on Functional Programming*, Uppsala, Sweden, 2003. To appear.

[16] Greg Morrisett, Matthias Felleisen, and Robert Harper. Abstract models of memory management. In *FPCA95: Conference on Functional Programming Languages and Computer Architecture*, pages 66–77, La Jolla, CA, June 1995.

[17] 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.

[18] Martin Odersky and Konstantin Läufer. Putting type annotations to work. In *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*, pages 54–67, New York, N.Y., 1996.

[19] Simon Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, 2003.

[20] 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.

[21] Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.*, 1998.

[22] Bratin Saha, Valery Trifonov, and Zhong Shao. Fully reflexive intensional type analysis in type erasure semantics. In *Third ACM SIGPLAN Workshop on Types in Compilation*, Montreal, September 2000.

[23] Carsten Schürmann, Joëlle Despeyroux, and Frank Pfenning. Primitive recursion for higher-order abstract syntax. *Theoretical Computer Science*, 266(1–2):1–58, September 2001.

[24] Tim Sheard. Type parametric programming. Technical Report CSE 93-018, Oregon Graduate Institute, 1993.

[25] Chris Stone and Robert Harper. Deciding type equivalence in a language with singleton kinds. In *Twenty-Seventh ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 214–225, Boston, MA, USA, January 2000.

[26] Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, September 2000.

[27] Philip Wadler. Theorems for free! In *FPCA89: Conference on Functional Programming Languages and Computer Architecture*, London, September 1989.

[28] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 60–76. ACM Press, 1989.

[29] Geoffrey Washburn and Stephanie Weirich. Boxes go bananas: Parametric higher-order abstract syntax in System F. To appear in the International Conference on Functional Programming, August 2003.

[30] Stephanie Weirich. Encoding intensional type analysis. In D. Sands, editor, *10th European Symposium on Programming*, volume 2028 of *Lecture Notes in Computer Science*, pages 92–106. Springer, 2001.

[31] Stephanie Weirich. Higher-order intensional type analysis. In Daniel Le Métayer, editor, *11th European Symposium on Programming*, pages 98–114, Grenoble, France, 2002.

[32] Stephanie Weirich. *Programming With Types.* PhD thesis, Cornell University, August 2002.

[33] Andrew K. Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115:38–94, 1994.

A Correctness of Embedding

We call the LH language with the operational semantic of Figure 6 LK. Below we prove the correctness of the translation between LK and LHR.

A.1 Static correctness

The static correctness of this translation follows from a straightforward set of inductive arguments. To prove that the translation of a LK term is well-typed in LHR, we must show that the representation of a LK-constructor has the correct representation type.

Because we essentially have two versions of type representations—one for constructors that may have variables bound by an enclosing *typerec*, and one for constructors that are in other contexts, there are two lemmas about the type soundness of the representations.

In these two results, we must define two different translations of Δ to produce the context for the type representations variables. In the first case, the translation is specialized by a return type constructor. The type of each representation variable must be specialized to this constructor. In the second case, for those variables bound by a term-level type abstraction (Λ), the types of the representations must be polymorphic over the return type.

$$|\Delta, \alpha{:}\kappa|_c = |\Delta|_c, y_\alpha : \mathcal{T}[\![[\tau]\langle\alpha : \kappa\rangle]\!]$$

$$|\Delta, \alpha{:}\kappa| = |\Delta|, x_\alpha : \widehat{R}\langle\alpha : \kappa\rangle$$

In the following two lemmas, we show that the representation of a LK constructor τ is well-typed. The free variables of τ may be bound in many different situations. We let Δ_1 refer to all of those bound by enclosing term-level type abstractions (Λ), Δ_2 refer to variables bound by type level type abstractions (λ), and Δ_3 list variables bound by enclosing *typerec* expressions. This lemma establishes the static correctness for closed representations, when the constructor does not appear inside of a *typerec*:

Lemma A.1 *Let* $\Delta = \Delta_1, \Delta_2$. *If* $\Delta \vdash \tau : \kappa$ *and* $\Delta_1, \Delta_2 \vdash \tau' : \star \to \star$ *then*

$$\Delta_1\Delta_2; |\Delta_1|, |\Delta_2|_{\tau'} \vdash \mathcal{R}[\![\tau, (\Delta_2, \tau', \bullet)]\!] : \mathcal{T}[\![[R\tau']\langle\tau : \kappa\rangle]\!]$$

The second lemma handles the case when we add the Ψ component to the representation. We cannot generalize these two lemmas into one as we need to use the first lemma to prove the second.

Lemma A.2 *Let* $\Delta = \Delta_1, \Delta_2, \Delta_3$ *and* $\Psi = (\Delta_3, \mathcal{E}[\![\eta]\!]\rho, \mathcal{E}[\![\theta]\!])$. *If* $\Delta \vdash \tau : \kappa$ *and* $\Delta_1, \Delta_2 \vdash \tau' : \star \to \star$ *and* $\Delta_1, \Delta_2; \Gamma[\tau'] \vdash \Delta_3 \mid \eta \mid \rho$ *and* $\Delta_1, \Delta_2; \Gamma \vdash \theta(\oplus) : [\tau]\langle\oplus : \kappa_\oplus\rangle$ *for* $(\theta(\oplus) \in Dom\, \theta)$. *then*

$$\Delta_1\Delta_2; |\Delta_1|, |\Delta_2|_{\tau'} \vdash \mathcal{R}[\![\tau, (\Delta_2, \tau', \Psi)]\!] : \mathcal{T}[\![[R\tau']\langle\rho(\tau) : \kappa\rangle]\!]$$

Theorem A.3 (Static Correctness) *If* $\Delta; \Gamma \vdash e : \sigma$ *then* $\Delta; |\Delta|, \mathcal{T}[\![\Gamma]\!] \vdash \mathcal{E}[\![e]\!] : \mathcal{T}[\![\sigma]\!]$

A.2 Dynamic correctness

We will prove operational correctness up to the definition in Figure A.2 of equivalence of result terms. The symbol $\equiv_\mathcal{E}$ relates two LHR terms that differ only by type β-expansions.

$\boxed{\text{Type-}\beta}$

$$\frac{}{(\Lambda\beta{:}\star \to \star.e)[\tau] \equiv_\mathcal{E} e[\tau/\beta]}$$

$\boxed{\text{Symmetry}}$

$$\frac{e' \equiv_\mathcal{E} e}{e \equiv_\mathcal{E} e'}$$

$\boxed{\text{Congruence rules}}$

$$\frac{}{i \equiv_\mathcal{E} i} \qquad \frac{}{x \equiv_\mathcal{E} x} \qquad \frac{}{R_\oplus \equiv_\mathcal{E} R_\oplus}$$

$$\frac{e \equiv_\mathcal{E} e'}{\lambda x{:}\sigma.e \equiv_\mathcal{E} \lambda x{:}\sigma.e'} \qquad \frac{e_1 \equiv_\mathcal{E} e_1' \quad e_2 \equiv_\mathcal{E} e_2'}{e_1 e_2 \equiv_\mathcal{E} e_1' e_2'}$$

$$\frac{e \equiv_\mathcal{E} e'}{\Lambda\alpha{:}\kappa.e \equiv_\mathcal{E} \Lambda\alpha{:}\kappa.e'} \qquad \frac{e \equiv_\mathcal{E} e'}{e[\tau] \equiv_\mathcal{E} e'[\tau]}$$

$$\frac{e \equiv_\mathcal{E} e' \qquad \theta(\oplus) \equiv_\mathcal{E} e_\oplus'}{\mathit{typerec}[\kappa][\tau]\ e\ \theta \equiv_\mathcal{E} \mathit{typerec}[\kappa][\tau]\ e'\ \theta'}$$

$$\frac{e \equiv_\mathcal{E} e' \qquad \theta(\oplus) \equiv_\mathcal{E} e_\oplus'}{\mathit{untyrec}[\kappa][\tau]\ e\ \theta \equiv_\mathcal{E} \mathit{untyrec}[\kappa][\tau]\ e'\ \theta'}$$

Figure 15: Type β-equivalence

This notion of equivalence does not weaken our dynamic-correctness result as all equal terms differ only in the type annotations. All equivalent terms have the same erasure, so we can argue that they model the same computation.

The reason that we can prove operational correctness only up to this notion of equivalence is because of how substitution interacts with the definition of representation. We would like substitution to commute with representation, but that is not the case.

$$\mathcal{R}[\![\tau_1[\tau_2/\alpha], (\Delta, \tau, \bullet)]\!] \neq \mathcal{R}[\![\tau_1, (\Delta, \tau, \bullet)]\!][\tau_2/\alpha][\widehat{\mathcal{R}}[\![\tau_2]\!]/x_\alpha]$$

For example, if τ_1 is α then the left hand side equals $\mathcal{R}[\![\tau_2, (\Delta, \tau, \bullet)]\!]$ while the right hand side equals $(x_\alpha[\tau])[\widehat{\mathcal{R}}[\![\tau_2]\!]/x_\alpha] = (\Lambda\beta{:}\star \to \star.\mathcal{R}[\![\tau_2, (\Delta, \beta, \bullet)]\!])[\tau]$.

Proposition A.4 *By examination of the definition of* $\equiv_\mathcal{E}$, *we assert the following properties of this relation:*

1. $\equiv_\mathcal{E}$ *is an equivalence relation.*
2. *If* $e_1 \equiv_\mathcal{E} e_2$ *then* $e[e_1/x] \equiv_\mathcal{E} e[e_2/x]$.
3. *If* $e_1 \equiv_\mathcal{E} e_2$ *then* $e_1[e/x] \equiv_\mathcal{E} e_2[e/x]$.
4. *If* e *is not of the form* $(\Lambda\beta{:}\star \to \star.e_1)[\tau]$ *and* $e \equiv_\mathcal{E} e'$ *then* $e' \mapsto^* e''$ *where* e'' *has the same outermost form as* e *and* $e'' \equiv_\mathcal{E} e$.

Lemma A.5 (Strengthening) *If* α *is not free in* τ, *then for any* Δ, c, τ', Ψ,

$$\mathcal{R}[\![\tau, (\Delta\{\alpha \Rightarrow \kappa\}, \tau', \Psi)]\!] = \mathcal{R}[\![\tau, (\Delta, \tau', \Psi)]\!]$$

Proof

Examination of the definition of $\mathcal{R}[\![\tau, \varepsilon]\!]$.

Lemma A.6 (Substitution of closed constructors) *If* $\Delta, \alpha{:}\kappa_2 \vdash \tau_1 : \kappa_1$ *and* $\emptyset \vdash \tau_2 : \kappa_2$ *then*

$$\mathcal{R}[\![\tau_1[\tau_2/\alpha], (\Delta, \tau, \bullet)]\!] \equiv_\mathcal{E} \mathcal{R}[\![\tau_1, (\Delta, \tau, \bullet)]\!][\tau_2/\alpha][\widehat{\mathcal{R}}[\![\tau_2]\!]/x_\alpha]$$

Lemma A.7 (Open substitution) *Let* $\Psi = (\Delta', \eta, \rho, \theta)$. *If* $\Delta, \alpha{:}\kappa' \vdash \tau_1 : \kappa$ *and* $\Delta \vdash \tau_2 : \kappa'$ *then*

$\mathcal{R}[\![\tau_1[\tau_2/\alpha], (\Delta, \tau, \Psi)]\!] \equiv_{\mathcal{E}}$
$\mathcal{R}[\![\tau_1, (\Delta\{\alpha \Rightarrow \kappa'\}, \tau, \Psi)]\!][\rho(\tau_2)/\alpha][\widehat{\mathcal{R}}[\![\rho(\tau_2)]\!]x_\alpha][\mathcal{R}[\![\tau_2, (\Delta, \tau, \Psi)]\!]/y_\alpha]$

Lemma A.8 *If* $\Delta \vdash \tau_1 : \kappa$ *and* $\tau_1 \leadsto^{wh} \tau_2$ *then for all* $e_1 \equiv_{\mathcal{E}} \mathcal{R}[\![\tau_1, (\Delta, \tau', \Psi)]\!]$, $e_1 \mapsto^* e_2$ *and* $e_2 \equiv_{\mathcal{E}} \mathcal{R}[\![\tau_2, (\Delta, \tau', \Psi)]\!]$.

Corollary A.9 *If* τ *weak head normalizes to* p, *and* $e \equiv_{\mathcal{E}} \mathcal{R}[\![\tau, (\Delta, c, \Psi)]\!]$ *then* $e \mapsto^* p' \equiv_{\mathcal{E}} \mathcal{R}[\![p, (\Delta, \tau, \Psi)]\!]$.

Lemma A.10 (Path correctness) *If*
$\emptyset \vdash_\kappa$ *typerec* $[\Delta, \eta, \rho][\tau']\langle p : \star\rangle$ *of* $\theta : \sigma$
and typerec $[\Delta, \eta, \rho][\tau']\langle p : \star\rangle$ *of* $\theta \Rightarrow_k e$ *and* $\theta' \equiv_{\mathcal{E}} \mathcal{E}[\![\theta]\!]$ *and*
$p' \equiv_{\mathcal{E}} \mathcal{R}[\![p, (\emptyset, \tau', (\Delta, \mathcal{E}[\![\eta]\!], \rho, \mathcal{E}[\![\theta]\!]))]\!]$ *then*

$$\textit{typerec}[\tau']\ p'\ \theta' \Rightarrow_{LHR} e_2 \equiv_{\mathcal{E}} \mathcal{E}[\![e]\!].$$

Lemma A.11 (Typerec Correctness) *Let* $\Psi = \Delta, \eta, \rho$. *If* *typerec* $\Psi[\tau']\langle\tau : \kappa\rangle$ *of* $\theta \mapsto_k e$ *and* $e_1 \equiv_{\mathcal{E}} \mathcal{E}[\![\textit{typerec}\ \Psi[\tau']\langle\tau : \kappa\rangle\ \textit{of}\ \theta]\!]$ *then* $e_1 \mapsto^*_{LHR} e_2 \equiv_{\mathcal{E}} \mathcal{E}[\![e]\!]$.

Lemma A.12 (Constructor substitution) *If* $\Delta, \alpha{:}\kappa; \Gamma \vdash e : \sigma$ *and* $\Delta \vdash \tau : \kappa$, *then* $\mathcal{E}[\![e[\tau/\alpha]]\!] \equiv_{\mathcal{E}} \mathcal{E}[\![e]\!][\tau/\alpha][\widehat{\mathcal{R}}[\![\tau]\!]x_\alpha]$.

Lemma A.13 (Term substitution) *If* $\Delta, ; \Gamma, x : \sigma' \vdash e : \sigma$ *and* $\Delta; \Gamma \vdash e' : \sigma'$, *then* $\mathcal{E}[\![e[e'/x]]\!] = \mathcal{E}[\![e]\!][\mathcal{E}[\![e']\!]/x]$.

Lemma A.14 (Dynamic correctness) *If* $\emptyset \vdash e_1 : \sigma$ *and* $e_1 \mapsto_k e_2$ *then if* $e_1' \equiv_{\mathcal{E}} \mathcal{E}[\![e_1]\!]$, $e_1' \mapsto^*_{LHR} e_2' \equiv_{\mathcal{E}} \mathcal{E}[\![e_2]\!]$.