JFP **16** (6): 681–710, 2006.
doi:10.1017/S0956796806005879 First published online 28 February 2006 Printed in the United Kingdom

Type-safe run-time polytypic programming

STEPHANIE WEIRICH

Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104, USA

Abstract

Polytypic programming is a way of defining type-indexed operations, such as map, fold and zip, based on type information. *Run-time* polytypic programming allows that type information to be dynamically computed – this support is essential in modern programming languages that support separate compilation, first-class type abstraction, or polymorphic recursion. However, in previous work we defined run-time polytypic programming with a type-passing semantics. Although it is natural to define polytypic programs as operating over first-class types, such a semantics suffers from a number of drawbacks. This paper describes how to recast that work in a type-erasure semantics, where terms represent type information in a safe manner. 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 (Abadi *et al*., 1991; Abadi *et al*., 1995; Crary & Weirich, 1999; Dubois *et al*., 1995; Harper & Morrisett, 1995; Hinze, 2000; Jansson & Jeuring, 1997; Jay *et al*., 1998; Ruehr, 1998; Sheard, 1993; Trifonov *et al*., 2000; Wadler & Blott, 1989).

Many of these frameworks generate polytypic operations at compile time through a source-to-source translation determined by static type information. In contrast, *run-time polytypic programming* (also called higher-order intensional type analysis (Weirich, 2002a)) defines polytypic operations with run-time analysis of dynamic 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. Many polytypic

operations defined for these types require type information for the abstracted types. Therefore, to define these operations, the semantics of the programming language must provide this type information at run time.

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 (Reynolds, 1983; Wadler, 1989) 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 (Morrisett *et al.*, 1995; Morrisett & Harper, 1997) 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 (Weirich, 2001) shows how to encode first-order run-time type analysis in F_ω (Girard, 1971) and Cheney and Hinze (Cheney & Hinze, 2002) implement the same capabilities in the Haskell language (Peyton Jones, 2003).

In *first-order* 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* 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 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 (2002) (CWM) describe how to support first-order 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, although 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 4, we cannot define a term that operates over these type constructor representations in the same way

Table 1. *Language comparison*

		Type analysis	Semantics
λ_R	(Crary *et al.*, 2002)	First-order	Type-erasure
LH	(Weirich, 2002a)	Higher-order	Type-passing
LHR	This paper	Higher-order	Type-erasure

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 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 type analysis and LHR, with a proof of correctness.
- A prototype implementation of LHR as a Haskell library that is simple, yet 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 3 reviews higher-order type analysis (formalized with the language LH) and section 4 discusses the problems with defining a type-erasure version of this language. In section 5 we present the type-erasure language called LHR. We describe the translation between LH and LHR in section 6. Section 7 describes the prototype implementation of LHR as a Haskell library. In section 8 we discuss extensions of this translation, and in section 9 we present related work and conclude. Appendix A contains the proof of correctness of the translation.

2 First-order type analysis with *typerec*

As a gentle introduction, we start with a common, first-order example of how *typerec* works in a language with a type-passing semantics. The language that this example is written in is an explicitly-typed polymorphic lambda calculus, much like System F. However, unlike System F, type arguments cannot be erased prior to execution as they are necessary for the *typerec* operator.

Figure 1 contains the archetypical *tostring* example that demonstrates the use of type analysis to automatically generate marshalling functions for any type. This functionality is reflected in the type of *tostring*: given any type α it returned a function that converts α's to strings. For example, the application $tostring[int \times int](3,4)$ returns the string "(3,4)".

The *typerec* operator works by folding over its argument type α. The $[\lambda\alpha.\alpha \rightarrow string]$ annotation is used for type checking. The branches θ tell what to do for

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{l}
tostring :: \forall\alpha{:}\star.\alpha \to string \\
tostring = \Lambda\alpha{:}\star.typerec[\lambda\alpha.\alpha \to string]\alpha\ of\ \ \theta \\
\\
\text{where } \theta = \\
\{\ \ int \ \ = \ \ string\_of\_int \\
\ \ \ unit \ \ = \ \ x{:}unit."()" \\
\ \ \ \times \ \ = \ \ \Lambda\beta{:}\star.\lambda x{:}(\beta \to string).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to string). \\
\qquad\qquad \lambda v{:}(\beta \times \gamma). \\
\qquad\qquad\quad "(" \ ++\ x(\pi_1 v) ++\ "," \ ++\ y(\pi_2 v) ++\ ")" \\
\ \ \ \to \ \ = \ \ \Lambda\beta{:}\star.\lambda x{:}(\beta \to string).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to string). \\
\qquad\qquad \lambda v : (\beta \to \gamma)."unprintable\ function" \\
\ \ \ + \ \ = \ \ \Lambda\beta{:}\star.\lambda x{:}(\beta \to string).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to string). \\
\qquad\qquad \lambda v : (\beta + \gamma).case\ v\ \ of \\
\qquad\qquad\quad (inj_1\ \ z \Rightarrow "inj1\ \ (" \ ++(xz)++\ ")" \\
\qquad\qquad\quad\ \ \mid inj_2\ \ z \Rightarrow "inj2\ \ (" \ ++(yz)++\ ")"\ ) \\
\}
\end{array}
$$

Fig. 1. Example: tostring.

each type constructor: For example, if α is *int* then a primitive function mapping integers to strings is returned. Some type constructors, such as $\times$, $\to$ and $+$ must be applied to arguments to form types. In the branches for those type constructors, *typerec* provides the arguments β and γ as well as marshallers for those types.

Unfortunately, the simple version of *typerec* described in this section is not expressive enough to define some polytypic operations, its argument must have base kind $\star$. In the next section, we discuss an extension of *typerec* that can analyze arguments of any kind. Furthermore, to be precise about the semantics of this extension, we completely specify the LH language that contains it.

3 LH: Higher-order analysis with type-passing

The LH language (see Figure 2) is a lightweight characterization of higher-order type analysis that captures the core ideas of the language of Weirich (Weirich, 2002a). It is a call-by-name variant of the Girard-Reynolds polymorphic lambda calculus (Girard, 1972; Girard, 1971; Reynolds, 1983) 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.

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

[1] Unlike other languages with intensional type analysis such as λ_i^{ML} (Harper & Morrisett, 1995) and λ_R (Crary *et al.*, 2002), LH does not include *Typerec*—a type constructor that defines other *types* by intensional analysis.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{ll}
\textit{(kinds)} & \kappa ::= \star \mid \kappa_1 \to \kappa_2 \\
\textit{(operators)} & \oplus ::= \mathit{int} \mid \to \mid \forall_\star \\
\textit{(type constructors)} & \tau ::= \alpha \mid \lambda\alpha{:}\kappa.\tau \mid \tau_1\tau_2 \mid \oplus \\
\textit{(types)} & \sigma ::= \tau \mid \mathit{int} \mid \sigma_1 \to \sigma_2 \mid \forall\alpha{:}\kappa.\sigma \\
\textit{(terms)} & e ::= i \mid x \mid \lambda x{:}\sigma.e \mid e_1 e_2 \\
 & \quad \mid \Lambda\alpha{:}\kappa.e \mid e[\tau] \\
 & \quad \mid \mathit{typerec}[\tau']\langle\tau : \kappa\rangle \mathit{\ of\ } \theta, \eta \\
 & \\
\textit{(typerec branches)} & \theta ::= \emptyset \mid \theta\{\oplus \Rightarrow e\} \\
\textit{(term environment)} & \eta ::= \emptyset \mid \eta\{\alpha \Rightarrow (\kappa, \tau, e)\} \\
\textit{(tycon context)} & \Delta ::= \emptyset \mid \Delta\{\alpha \Rightarrow \kappa\} \\
\textit{(term context)} & \Gamma ::= \emptyset \mid \Gamma\{x \Rightarrow \sigma\} \\
\textit{(operator signature)} & \Sigma ::= \{\ \mathit{int} \Rightarrow \star, \\
 & \qquad\qquad \to \Rightarrow \star \to \star \to \star, \\
 & \qquad\qquad \forall_\star \Rightarrow (\star \to \star) \to \star\}
\end{array}
$$

Fig. 2. Syntax of LH.

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 θ, η, etc. We write the empty map as $\emptyset$, add a new binding to θ with $\theta\{\oplus \Rightarrow e\}$ (defined only when $\oplus \notin \mathit{Dom}(\theta)$) and retrieve a binding with $\theta(\oplus)$ (defined only when $\oplus \in \mathit{Dom}(\theta)$). The notation for the other maps is analogous.

The term $\mathit{typerec}[\tau']\langle\tau : \kappa\rangle \mathit{\ of\ } \theta, \eta$ 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. This environment maps a type variable α to a triple (κ, τ, e), describing the kind of α, a substitution for α when it appears outside the scope of the *typerec*, and its interpretation. We use the notation $\Delta(\eta)$ to create a type context from the variables bound by η and the notation $\eta(\tau)$ to substitute for those type variables that appear free in the type constructor τ.

In a *typerec* term, the type constructor τ' is an annotation that makes type checking syntax directed. We call it the *return type constructor* and usually use the metavariable τ' to refer to it. The return type constructor is used to determined the type of an analysis of a type constructor τ of kind κ, defined to be $[\tau']\langle\eta(\tau) : \kappa\rangle$ using the definition of a *polykinded type*, below.

[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.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{l}
size = \\
\quad \Lambda\alpha{:}\star \to \star.\, typerec\,[\lambda\beta{:}\star.\beta \to int]\langle\alpha : \star \to \star\rangle\ of\ \theta, \emptyset \\
\\
\text{where } \theta = \\
\quad \{\ int \quad \Rightarrow \lambda y{:}int\ .0 \\
\quad\ \ unit \quad \Rightarrow \lambda y{:}unit\ .0 \\
\quad\ \ \times \quad \Rightarrow \Lambda\beta{:}\star.\lambda x{:}(\beta \to int).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to int). \\
\qquad\qquad \lambda v{:}(\beta \times \gamma). \\
\qquad\qquad\quad x(\pi_1 v) + y(\pi_2 v) \\
\quad\ \ \to \quad \Rightarrow undefined \\
\quad\ \ + \quad \Rightarrow \Lambda\beta{:}\star.\lambda x{:}(\beta \to int).\Lambda\gamma{:}\star.\lambda y{:}(\gamma \to int). \\
\qquad\qquad \lambda v{:}(\beta + \gamma).\ case\ \ v\ \ of \\
\qquad\qquad\quad (inj_1\, z \Rightarrow x(z)\ \mid\ inj_2\, z \Rightarrow y(z)) \\
\quad\ \ \forall_\star \quad \Rightarrow undefined \\
\quad\ \ \exists_\star \quad \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}(\forall\beta{:}\star.(\beta \to int) \to \alpha\beta \to int). \\
\qquad\qquad \lambda x{:}\exists\alpha. \\
\qquad\qquad\quad let\ \langle\beta, y\rangle = unpack\ x\ in \\
\qquad\qquad\qquad (r\ [\beta]\ (\lambda x{:}\beta.0)\ y) \\
\quad\ \ \mu_\star \quad \Rightarrow \Lambda\alpha{:}\star \to \star. \\
\qquad\qquad \lambda x{:}(\forall\beta{:}\star.(\beta \to int) \to \alpha\beta \to int). \\
\qquad\qquad\quad fix\ f{:}(\mu_\star\alpha) \to int\ . \\
\qquad\qquad\qquad \lambda y{:}\mu_\star\alpha.\ (x\ [\mu_\star\alpha]\ f\ (unroll\ y)) \\
\quad \}
\end{array}
$$

Fig. 3. Example: *size*.

Definition 3.1

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:

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

A simple example of a *typerec* term is

$$
typerec\ [\lambda\beta{:}\star.\beta]\langle\alpha : \star\rangle\ of\ \ \{int \Rightarrow 0\}, \{\alpha \Rightarrow (\star, int, 3)\}
$$

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

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

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

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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$\boxed{\Delta;\Gamma[\tau'] \vdash \eta}$$

$$\frac{}{\Delta;\Gamma[\tau'] \vdash \emptyset}$$

$$\frac{\Delta;\Gamma[\tau'] \vdash \eta \qquad \Delta;\Gamma \vdash e : [\tau']\langle \tau : \kappa \rangle \qquad \Delta \vdash \tau : \kappa \qquad \beta \notin Dom(\Delta, \eta)}{\Delta;\Gamma[\tau'] \vdash \eta\{\beta \Rightarrow (\kappa, \tau, e)\}}$$

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

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

Fig. 4. Static semantics of LH *typerec*.

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

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

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, $length = \Lambda\beta{:}\star.size[List][\beta](\lambda x{:}\beta.1)$. 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 $Tree \stackrel{\text{def}}{=} \lambda\beta{:}\star.\mu_\star(\lambda\alpha : \star.\beta + (\alpha \times \alpha))$ or $Maybe \stackrel{\text{def}}{=} \lambda\beta : \star.unit + \beta$.

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.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

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

$$\frac{}{typerec\,[\tau']\langle (\lambda\alpha{:}\kappa'.\tau_1) : \kappa' \to \kappa \rangle\ of\ \theta, \eta \mapsto_h \Lambda\beta{:}\kappa'.\lambda x{:}[\tau']\langle \beta : \kappa' \rangle.\ typerec\,[\tau']\langle \tau_1 : \kappa \rangle\ of\ \theta, \eta\{\alpha \Rightarrow (\kappa', \beta, x)\}}$$

$$\frac{\Delta(\eta) \vdash \tau_2 : \kappa'}{typerec\,[\tau']\langle \tau_1 \tau_2 : \kappa \rangle\ of\ \theta, \eta \mapsto_h (typerec\,[\tau']\langle \tau_1 : \kappa' \to \kappa \rangle\ of\ \theta, \eta)\ [\eta(\tau_2)]\ (typerec\,[\tau']\langle \tau_2 : \kappa' \rangle\ of\ \theta, \eta)}$$

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

Fig. 5. Dynamic semantics of LH *typerec*.

In the expression $typerec\,[\tau']\langle \tau : \kappa \rangle\ of\ \theta, \eta$, the environment η used to interpret those type variables. We check that η is well-formed with the judgment form $\Delta; \Gamma[\tau'] \vdash \eta$. This judgment declares that η maps type variables to appropriate terms for the return type constructor $[\tau']$, and to types of the appropriate kind. 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 return type constructor is well formed, the environment is well formed, the argument τ is well formed (with context extended by the variables bound in the environment, $\Delta(\eta)$) 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 η 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 $\lambda\alpha{:}unit + \alpha$. 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 $size[Maybe][unit](\lambda x : unit\ .1)$ does so for arguments of type $Maybe\,unit$. We can trace the evaluation of this term as follows. Let η be the environment $\{\alpha \Rightarrow \star, unit, (\lambda x{:}unit\ .1)\}$, let θ be the branches for *size* and let τ' be the return type constructor $(\lambda\alpha{:}\star\ .\alpha \to int))$:

$$\begin{aligned} &size[\lambda\alpha{:}\star\ .\alpha + unit][unit](\lambda x{:}unit\ .1) \\ &\quad \mapsto_h (\Lambda\beta{:}\star\ .\lambda w{:}(\alpha \to int).\lambda v{:}(\alpha \times unit). \\ &\qquad typerec\,[\tau']\langle \alpha + unit\ : \star \rangle \\ &\qquad\quad of\ \ \theta, \{\alpha \Rightarrow \star, \beta, w\})\ [unit](\lambda x{:}unit\ .1) \end{aligned}$$

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{l}
\mapsto_h typerec[\tau']\langle\alpha + unit : \star\rangle\ of\ \ \theta, \eta \\
\mapsto_h (typerec[\tau']\langle + : \star \to \star \to \star\rangle) \\
\quad [unit](typerec[\tau']\langle\alpha : \star\rangle\ of\ \ \theta, \eta) \\
\quad [unit](typerec[\tau']\langle unit : \star\rangle\ \ of\ \ \theta, \eta) \\
\mapsto_h (\Lambda\beta : \star .\lambda x : (\beta \to int).\Lambda\gamma : \star .\lambda y : (\gamma \to int). \\
\quad \lambda v : (\beta + \gamma).\ case\ \ v\ \ of \\
\qquad (inj_1\, z \Rightarrow x(z) \mid\ inj_2\, z \Rightarrow y(z))) \\
\quad [unit](typerec[\tau']\langle\alpha : \star\rangle\ \ of\ \ \ \theta, \eta) \\
\quad [unit](typerec[\tau']\langle unit : \star\rangle\ of\ \ \ \theta, \eta) \\
\mapsto_h \lambda v : (unit + unit).\, case\ v\ \ of \\
\quad (inj_1\, z \Rightarrow (typerec[\tau']\langle\alpha : \star\rangle\ \ of\ \ \ \theta, \eta)(z) \\
\quad \mid\ inj_2\, z \Rightarrow (typerec[\tau']\langle unit : \star\rangle\ \ of\ \ \ \theta, \eta)(z))
\end{array}
$$

Reduction shows that this result is equivalent to: $\lambda v : (unit + unit).\, case\ v\ of\ (inj_1\, z \Rightarrow 1 \mid\ inj_2\, z \Rightarrow 0)$

4 The problem with type constructor representations

The LH language requires a type-passing semantics. The operational 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 (2002) (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 as the basis of 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 $R\ int$, 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, $\mathscr{R}[\![\tau]\!]$, the representation of the type

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

τ is defined as:

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

The type of a representation term is determined by the kind of the constructor it represents. If τ has kind κ, then $\mathscr{R}[\![\tau]\!]$ has the polykinded type $[R]\langle\tau : \kappa\rangle$. However, because other terms besides $\mathscr{R}[\![\tau]\!]$ have type $[R]\langle\tau : \kappa\rangle$, it is difficult to define an operational semantics for *typerec* based on matching $\mathscr{R}[\![\tau]\!]$. 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 Weirich (2002b) 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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

Paths

$$\rho ::= \alpha \mid \oplus \mid \rho\ \tau$$

$\tau \rightsquigarrow^{wh} \rho$

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

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

$e \Rightarrow_k e'$

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

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

$e \mapsto_k e'$

$$\frac{\tau \rightsquigarrow^{wh} \rho \qquad \mathit{typerec}[\tau']\langle\rho : \star\rangle\ \mathit{of}\ \theta, \eta \Rightarrow_k e}{\mathit{typerec}[\tau']\langle\tau : \star\rangle\ \mathit{of}\ \theta, \eta \mapsto_k e}$$

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

Fig. 6. Kind-directed operational semantics.

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.

Instead, there is a simpler way to define the type-erasure calculus, based on an implementation of induction over higher-order abstract syntax (Fegaras & Sheard, 1996; Washburn & Weirich, 2003). 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.

5 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 $\star$, but as we describe below, that will is not a limitation to its expressiveness.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

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

Fig. 7. Syntax of LHR.

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

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

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

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

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

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

Fig. 8. LHR: Operational semantics of *typerec*.

Each rule for a specific operator of LHR in Figure 8 is generated from the following general rule that corresponds to $\Rightarrow_k$ evaluation of an operator of kind $\kappa_1 \to \ldots \to \kappa_n \to \star$.

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

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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

LHR Polykinded types

$$[\tau']\langle\langle \tau : \star\rangle\rangle \stackrel{\text{def}}{=} \tau'\tau$$
$$[\tau']\langle\langle \tau : \kappa_1 \to \kappa_2\rangle\rangle \stackrel{\text{def}}{=} \forall\alpha{:}\kappa_1.\widehat{R}\langle\alpha : \kappa_1\rangle \to [\tau']\langle\langle \alpha : \kappa_1\rangle\rangle \to [\tau']\langle\langle \tau\alpha : \kappa_2\rangle\rangle$$

$$\widehat{R}\langle\tau : \kappa\rangle \stackrel{\text{def}}{=} \forall\alpha{:}\kappa.[R\alpha]\langle\langle \tau : \kappa\rangle\rangle$$

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

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

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

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

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

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

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

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

$$\frac{\Delta \vdash \tau : \star \qquad \Delta \vdash \tau' : \star \to \star \qquad \Delta;\Gamma \vdash e : \tau'\tau}{\Delta;\Gamma \vdash untyrec[\tau']\ e\ \ : R\ \tau'\tau}$$

Fig. 9. LHR: Static Semantics (excerpt)

variables. Where we might analyze an argument with a free type variable in LH:

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

we will translate that term to the LHR term:

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

Figure 9 shows the static semantics for the representation terms, *typerec* and *untyrec*. 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.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{rcl}
typerec[\tau']\langle e_\tau : \star\rangle of\ \theta & \stackrel{\text{def}}{=} & typerec[\tau']\ e_\tau\ of\ \theta \\
typerec[\tau']\langle e_\tau : \kappa_1 \to \kappa_2\rangle of\ \theta & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa_1.\lambda x{:}\widehat{R}\langle\alpha : \kappa_1\rangle.\lambda y{:}[\tau']\langle\langle\alpha : \kappa_1\rangle\rangle.N \\
 & & \text{where} \\
 & & N = typerec[\tau']\langle(e_\tau[\alpha]\ x\ M) : \kappa_2\rangle of\ \theta \\
 & & M = untyrec[\tau']\langle y : \kappa_1\rangle of\ \theta \\
 & & \\
untyrec[\tau']\langle e_\tau : \star\rangle of\ \theta & \stackrel{\text{def}}{=} & untyrec[\tau']\ e_\tau \\
untyrec[\tau']\langle e_\tau : \kappa_1 \to \kappa_2\rangle of\ \theta & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa_1.\lambda x{:}\widehat{R}\langle\alpha : \kappa_1\rangle.\lambda y{:}[\tau']\langle\langle\alpha : \kappa_1\rangle\rangle.N \\
 & & \text{where} \\
 & & N = untyrec[\tau']\langle(e_\tau[\alpha]\ x\ M) : \kappa_2\rangle of\ \theta \\
 & & M = typerec[\tau']\langle y : \kappa_1\rangle of\ \theta
\end{array}
$$

Fig. 10. Higher-order *typerec* in LHR.

When a term representation is polymorphic over this return type constructor (for example, if it is of type $\forall\beta{:}\star \to \star.R\beta\tau$) 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.

The notation $\widehat{R}\langle\tau : \kappa\rangle$ is also used for the type of the representation constants. If $\oplus$ is an arbitrary type constructor constant, such as *int*, $\to$, $\forall_\star$ 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 type $\widehat{R}\langle\tau : \kappa\rangle$ is mutually defined in terms of the LHR definition of polykinded types, $[\tau']\langle\langle\tau : \kappa\rangle\rangle$ (at the top of Figure 9). The difference between these polykinded types and those of LH is the additional representation argument. LHR polykinded types are also used to type the branches of *typerec*. That way, branches such as $\times$ or $+$ receive the representation of their type arguments.

A similar change is to the type equivalence rule for the operator $\forall_\star$. (If $\exists_\star$ were in our formal language, we would define its equivalence rule similarly.) Some polytypic functions defined over quantified types need to analyze the hidden type. By changing the type equivalence rule in this way, we make sure the representation of that bound variable is accessible for analysis. It is also possible to add the operator $\hat{\forall}_\star$ to this calculus, such that $\hat{\forall}_\star\tau = \forall\alpha{:}\star.\tau\alpha$. This operator produces the type of parametric functions that cannot analyze their type arguments. This operator makes no sense in LH, because all types are analyzable.

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

$$typerec[\tau']\langle List : \star \to \star\rangle\ of\ \theta, \eta$$

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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

interpretation.

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

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 $typerec[\tau']\langle e_\tau : \kappa\rangle\ of\ \theta$, is in Figure 10. 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.

This completes the description of LHR. As is standard, we have shown that the static semantics agrees with the dynamic semantics.

Theorem 5.1 (*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. □

In the next section, we will show that LHR is as expressive as LH by defining a translation between the two languages. To gain an intuition behind this translation, we end this section with an example in Figure 11, *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 the *List* type constructor.

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

There are two key 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 $\widehat{\mathscr{R}}[\![\mu_\star\alpha]\!]$ defined in the next section.)

6 Translating LH to LHR

The translation between LH and LHR is based on a process called 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 $\mathscr{T}[\![\sigma]\!]$ and terms $\mathscr{E}[\![e]\!]$ appears in Figure 13. Kinds and type constructors are unchanged. We use a number of auxiliary definitions in this translation, listed in Figure 12. An invariant of this translation is that whenever a type variable, α, is in scope, its term representation is also in scope as variable x_α. Therefore, every type abstraction ($\Lambda\alpha$) is immediately followed by an abstraction of its representation (λx_α). Consequently, the type translation for polymorphic types includes the type of this additional argument. When a polymorphic term is applied to a type argument τ, that argument is followed by its term representation, $\widehat{\mathscr{R}}[\![\tau]\!]$.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{l}
size = \Lambda\alpha{:}\star\,.\lambda x_\alpha{:}R(\lambda\beta{:}\star\,.\beta \to int)\,\alpha. \\
\quad typerec[\lambda\beta{:}\star\,.\beta \to int]\ x_\alpha\ of\ \ \theta \\
\\
\text{where } \theta = \\
\{\ int \quad \Rightarrow \lambda y{:}int\,.0 \\
\quad unit \quad \Rightarrow \lambda y{:}unit\,.0 \\
\quad \times \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda x_\beta{:}\widehat{R}\langle\beta\ {:}\star\rangle.\lambda x{:}(\beta \to int). \\
\qquad\qquad \Lambda\gamma{:}\star\,.\lambda x_\gamma{:}\widehat{R}\langle\gamma\ {:}\star\rangle.\lambda y{:}(\gamma \to int). \\
\qquad\qquad\quad \lambda v{:}(\beta \times \gamma). \\
\qquad\qquad\qquad x(\pi_1 v) + y(\pi_2 v) \\
\quad \to \quad \Rightarrow undefined \\
\quad + \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda x_\beta{:}\widehat{R}\langle\beta\ {:}\star\rangle.\lambda x{:}(\beta \to int). \\
\qquad\qquad \Lambda\gamma{:}\star\,.\lambda x_\beta{:}\widehat{R}\langle\gamma\ {:}\star\rangle.\lambda y{:}(\gamma \to int). \\
\qquad\qquad\quad \lambda v{:}(\beta + \gamma).\ case\ \ v\ \ of \\
\qquad\qquad\qquad (inj_1\, z \Rightarrow x(z)\ \mid\ inj_2\, z \Rightarrow y(z)) \\
\quad \forall_\star \quad \Rightarrow undefined \\
\quad \exists_\star \quad \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda x_\alpha{:}\widehat{R}\langle\alpha\ {:}\star \to \star\rangle. \\
\qquad\qquad \lambda r{:}(\forall\beta{:}\star\,.\widehat{R}\langle\beta\ {:}\star\rangle \to (\beta \to int) \to \alpha\beta \to int). \\
\qquad\qquad\quad \lambda x{:}(\exists\beta\ {:}\star.\widehat{R}\langle\beta\ {:}\star\rangle \times (\alpha\beta)). \\
\qquad\qquad\qquad let\ \langle\beta, \langle x_\beta, y\rangle\rangle = unpack\ x\ in \\
\qquad\qquad\qquad\quad (r\ \ [\beta]\ \ x_\beta\ \ (\lambda x{:}\beta.0)\ \ y) \\
\quad \mu_\star \quad \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda x_\alpha{:}\widehat{R}\langle\alpha\ {:}\star \to \star\rangle. \\
\qquad\qquad \lambda x{:}(\forall\beta{:}\star\,.\widehat{R}\langle\beta\ {:}\star\rangle \to (\beta \to int) \to \alpha\beta \to int). \\
\qquad\qquad\quad fix\ f{:}(\mu_\star\alpha \to int). \\
\qquad\qquad\qquad \lambda y{:}\mu_\star\alpha.\ (x\ \ [\mu_\star\alpha]\ \ \widehat{\mathscr{R}}[\![\mu_\star\alpha]\!]\ \ f\ \ (unroll\ y)) \\
\quad \}
\end{array}
$$

Fig. 11. Example: Erasure version of *size*.

Translation from LH to LHR		
Type translation	$\mathscr{T}[\![\sigma]\!]$	(Figure 13)
Term translation	$\mathscr{E}[\![e]\!]$	(Figure 13)
Derived forms in LHR		
LHR Polykinded type	$[\tau']\langle\langle\tau\ {:}\ \kappa\rangle\rangle$	(Figure 9)
Higher-order *typerec*	$typerec[\tau']\langle e\ {:}\ \kappa\rangle of\ \ \theta$	(Figure 10)
Higher-order *untyrec*	$untyrec[\tau']\langle e\ {:}\ \kappa\rangle of\ \ \theta$	(Figure 10)
General type representation	$\widehat{\mathscr{R}}[\![\tau]\!]$	(Figure 14)
Specialized representation	$\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')}$	(Figure 14)

Fig. 12. Notation used in the translation.

The most important part of this translation is the translation of *typerec*, at the bottom of Figure 13. This translation replaces the argument to *typerec*, not with the standard representation of the type argument, but one that is specialized to the *typerec* term. The appropriate *typerec* argument is constructed in two phases: First a "specialized representation" $\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')}$ is constructed, which may contains references

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{aligned}
\mathscr{T}[\![\tau]\!] &= \tau \\
\mathscr{T}[\![int]\!] &= int \\
\mathscr{T}[\![\sigma_1 \to \sigma_2]\!] &= \mathscr{T}[\![\sigma_1]\!] \to \mathscr{T}[\![\sigma_2]\!] \\
\mathscr{T}[\![\forall\alpha{:}\kappa.\sigma]\!] &= \forall\alpha{:}\kappa.\widehat{R}\langle\alpha : \kappa\rangle \to \mathscr{T}[\![\sigma]\!] \\
\mathscr{T}[\![R\tau\tau']\!] &= R\ \tau\ \tau'
\end{aligned}
$$

$$
\begin{aligned}
\mathscr{E}[\![i]\!] &= i \\
\mathscr{E}[\![\lambda x{:}\sigma.e]\!] &= \lambda x{:}\mathscr{T}[\![\sigma]\!].\mathscr{E}[\![e]\!] \\
\mathscr{E}[\![e_1 e_2]\!] &= \mathscr{E}[\![e_1]\!]\mathscr{E}[\![e_2]\!] \\
\mathscr{E}[\![\Lambda\alpha{:}\kappa.e]\!] &= \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.\mathscr{E}[\![e]\!] \\
\mathscr{E}[\![e[\tau]]\!] &= \mathscr{E}[\![e]\!]\ [\tau]\ \widehat{\mathscr{R}}[\![\tau]\!] \\
\mathscr{E}[\![\ typerec[\tau']\langle\tau : \kappa\rangle\ of\ \ \theta, \eta\ \ ]\!] &= typerec[\tau']\langle\Phi(\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')}) : \kappa\rangle of\ \mathscr{E}[\![\theta]\!] \\
&\quad \text{where for each } \{\alpha \Rightarrow (\kappa_\alpha, \tau_\alpha, e)\} \in \eta \\
&\quad\ \Delta(\alpha) = \kappa_\alpha \\
&\quad\ \Phi(\alpha) = \tau_\alpha \\
&\quad\ \Phi(x_\alpha) = \widehat{\mathscr{R}}[\![\tau_\alpha]\!] \\
&\quad\ \Phi(y_\alpha) = untyrec[\tau']\langle\mathscr{E}[\![e]\!] : \kappa_\alpha\rangle\ \ of\ \ \ \mathscr{E}[\![\theta]\!]
\end{aligned}
$$

Fig. 13. Translation of LH to LHR.

to the term variables x_α and y_α for each α in the domain of the environment η. We describe this process in the next section—briefly, the x_α indicate where the representation of the type τ_α is needed, and the y_α mark where *untyrec* should embed the branch for α. After the construction of the specialized representation, all occurrences of α, x_α and y_α are replaced by the substitution Φ.

6.1 Representing the constructor language

The definition of type representations ($\widehat{\mathscr{R}}[\![\tau]\!]$) and specialized representations ($\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')}$) is in Figure 14. Type representations are defined in terms of specialized representations at the bottom of the figure.

Specialized representations are used for the argument to *typerec*. These representation are specialized to τ' the return type constructor of an analysis of this term, and to Δ, a context containing type variables. As before, type variables are represented by term variables, but here, each type variable has both a "specialized representation", y_α, of type $[R\tau']\langle\langle\alpha : \kappa\rangle\rangle$ as well as its standard representation, x_α, of type $\widehat{R}\langle\alpha : \kappa\rangle$). The context Δ determines which term variable should be used, the specialized representation or an instantiation of the general one.

Most of the time, the specialized representation should be used. To enable this, type-level abstractions are translated to abstractions that provide not just the general representation of a type argument, but its specialized representation as well.

A general type representation (defined in the last line of the figure) abstracts over the return type constructor, so that it may be used in any analysis. Because the type of the y_α depend on the return type constructor, they should not be used for type variables that are currently in scope. Instead, a general type representation uses the empty context, and represents any currently in scope type variables with $x_\alpha[\tau']$,

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

$$
\begin{array}{rcl}
\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')} & : & [R\,\tau']\langle\langle \tau : \kappa \rangle\rangle \\
\\
\mathscr{R}[\![\oplus]\!]_{(\Delta,\tau')} & \stackrel{\text{def}}{=} & R_{\oplus}[\tau'] \\
\mathscr{R}[\![\alpha]\!]_{(\Delta,\tau')} & \stackrel{\text{def}}{=} & \begin{cases} y_\alpha & \text{if } \alpha \in \mathit{Dom}\,\Delta \\ x_\alpha[\tau'] & \text{otherwise} \end{cases} \\
\mathscr{R}[\![\lambda\alpha{:}\kappa.\tau_1]\!]_{(\Delta,\tau')} & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\kappa.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \kappa\rangle.\lambda y_\alpha{:}[R\,\tau']\langle\langle\alpha : \kappa\rangle\rangle.\mathscr{R}[\![\tau_1]\!]_{(\Delta\{\alpha\Rightarrow\kappa\},\tau')} \\
\mathscr{R}[\![\tau_1\,\tau_2]\!]_{(\Delta,\tau')} & \stackrel{\text{def}}{=} & \mathscr{R}[\![\tau_1]\!]_{(\Delta,\tau')}\ [\tau_2]\ \widehat{\mathscr{R}}[\![\tau_2]\!]\ \mathscr{R}[\![\tau_2]\!]_{(\Delta,\tau')} \\
\\
\widehat{\mathscr{R}}[\![\tau]\!] & : & \widehat{R}\langle\tau : \kappa\rangle = \forall\alpha{:}\star \to \star.[R\alpha]\langle\langle\tau : \kappa\rangle\rangle \\
\\
\widehat{\mathscr{R}}[\![\tau]\!] & \stackrel{\text{def}}{=} & \Lambda\alpha{:}\star \to \star.\mathscr{R}[\![\tau]\!]_{(\emptyset,\alpha)}
\end{array}
$$

Fig. 14. Representation of LHR type constructors.

the general representation for that type variable instantiated with the return type constructor.

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

$$
\begin{array}{l}
\Lambda\beta{:}\star \to \star.\Lambda\alpha{:}\star.\lambda x_\alpha{:}\widehat{R}\langle\alpha : \star\rangle.\lambda y_\alpha{:}[R\beta]\langle\langle\alpha : \star\rangle\rangle. \\
\quad R_{\to}[\beta]\ [\alpha]\ (\Lambda\gamma{:}\star \to \star.x_\alpha[\gamma])\ y_\alpha\ [\mathit{int}]\ (\Lambda\gamma{:}\star \to \star.R_{\mathit{int}}[\gamma])\ (R_{\mathit{int}}[\beta])
\end{array}
$$

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

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

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

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

7 Implementation

In this section we describe an implementation of a simplified version of LHR to show how these ideas could be incorporated into a language like Haskell. For simplicity, our implementation is a Haskell library [3]. An extension to Haskell might

[3] This implementation requires the extensions of first-class polymorphism and existential types (Odersky & Läufer, 1996) supported by the implementations GHC and Hugs.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

be more attractive to programmers, but the important details of the implementation are present in this version.

The interface to this implementation is the following:

```
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.  c a ->  c b -> c (a, b),
  name  :: forall b. (String, [DC c b]) -> c b,
  ex    :: forall a. (forall b. c b -> 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]
```

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

```
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 general representation of the hidden type (i.e. $\exists a.(\forall c.\ R\ c\ a \times f\ a)$).

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

We could also omit the general 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 general 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) -> c a -> c a
          -> (forall c. R c b) -> c b -> c b
          -> c (a, b)
```

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

However, even without general representations, we have enough information to implement the size example. To pass the return type constructor ($\lambda\alpha{:}\star.\alpha \to int$) 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 11, 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`.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

```
                    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
            unRS (xa (typerec thetaShow 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 specific representation. For most flexibility in calling other polytypic functions, we need the general 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.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

For example, we can represent the type $\exists\alpha.\, int \times \alpha$ 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
        unRS (xa (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.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

```
rtimes :: R c a -> R c b -> R c (a, b)
rtimes t1 t2 = R (\theta -> times theta (typerec theta t1)
                (typerec theta t2))
rname  :: (String, [DC (R c) b]) -> R c b
rname  = \(str,cons) -> R (\x -> name x (str,cons))
rname  = \(str,cons) -> R (\x -> name x (str,loop x cons))
            where loop x [] = []
                  loop x (DC str rep inn out : rest) =
                      DC str (typerec x rep) inn out : loop x rest
rex    :: (forall b. R c b -> R c (a b)) -> R c (Ex a)
rex t1 = R (\x -> ex x (\y -> (typerec x (t1 (untyrec y)))))
```

8 Extensions

LH is only a subset of the language described by Weirich (2002a). 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 (Hinze, 2000)) 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 $typerec[\tau'](e : \kappa) of\ \theta$ and $untyrec[\tau'](e : \kappa) of\ \theta$ 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.

9 Summary and related work

This paper develops a type-erasure language supporting higher-order type analysis, necessary for run-time polytypic programming. While type-erasure versions of several

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

other type analyzing languages have been previously developed (Crary *et al.*, 2002; Saha *et al.*, 2000), 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* (Stone & Harper, 2000), which was inspired by Coquand's approach to $\beta\eta$-equivalence for a type theory with Π types and one universe (Coquand, 1991). 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.* (2000) who themselves refer to Fegaras & Sheard (1996). 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 & Weirich (2003) 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 & Pfenning (2001).

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 simple to implement: we give a prototype implementation in only a few lines. Closely related work to this paper is a proposal for Dependency-Style Generic Haskell (Löh *et al.*, 2003) that addresses the problem in Generic Haskell of defining polytypic operations that depend on one another. Because general 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.

Important future work is the integration of *type-level* type analysis to this language, as is found in intensional type. Although there are many examples of polytypism where the result of a type-analyzing function can be described parametrically in terms of its argument, this is not always the case. For example, Hinze *et al.* (2002) describe how the type of generalized tries depends on the key type.

Other future work includes a practical implementation based on the LHR language in this paper. The prototype implementation has the advantage of being small and implementable as a Haskell library. However, if these facilities were provided as a Haskell extension, they might be made more easy for programmers to use. For example, some of the complexity of defining polytypic operations comes from using a newtype to specify the return type constructor. A specialized extension could integrate some form of local type inference (Pierce & Turner, 1998) to specify this type constructor and not require the coercions to and from the newtype. Furthermore, defining the branches of the polytypic function as records is somewhat awkward, and could be improved with specialized syntax. Finally, a Haskell extension could automatically define the representations of user defined datatypes, instead of requiring that they be supplied by users.

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.

$$|\Delta, \alpha{:}\kappa|_c = |\Delta|_{\tau'}, y_\alpha : [\tau']\langle\langle \alpha : \kappa \rangle\rangle$$

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

In the following two lemmas, we show that the representation of a 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) or by enclosing *typerec* expressions.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

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

$$\Delta_1 \Delta_2 ; |\Delta_1|, |\Delta_2|_{\tau'} \vdash \mathscr{R}[\![\tau]\!]_{((\Delta_2, \tau'))} : [R\tau'] \langle\langle \tau : \kappa \rangle\rangle$$

Another lemma that we need is that the translation of a LH polykinded type is a LHR polykinded type.

Lemma A.2
If $\mathscr{T}[\![[\tau'] \langle \tau : \kappa \rangle]\!] = [\tau'] \langle\langle \tau : \kappa \rangle\rangle$.

We may now prove the static correctness of phase-splitting.

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

A.2 Dynamic correctness

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

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

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

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

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

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

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

Proof
Examination of the definition of $\mathscr{R}[\![\tau]\!]_{(\Delta,\tau')}$. □

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

Type-β

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

Symmetry

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

Congruence rules

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

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

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

$$\frac{e \equiv_{\mathscr{E}} e' \qquad \theta(\oplus) \equiv_{\mathscr{E}} e'_{\oplus}}{typerec[\kappa][\tau]\ e\ \theta \equiv_{\mathscr{E}} typerec[\kappa][\tau]\ e'\ \theta'}$$

$$\frac{e \equiv_{\mathscr{E}} e' \qquad \theta(\oplus) \equiv_{\mathscr{E}} e'_{\oplus}}{untyrec[\kappa][\tau]\ e\ \theta \equiv_{\mathscr{E}} untyrec[\kappa][\tau]\ e'\ \theta'}$$

Fig. A 1. Type β-equivalence.

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

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

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

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

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

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

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

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

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

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

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

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

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

References

Abadi, M., Cardelli, L., Pierce, B. and Plotkin, G. (1991) Dynamic typing in a statically-typed language. *ACM Trans. Program. Lang. Syst.* **13**(2): 237–268.

Abadi, M., Cardelli, L., Pierce, B. and Rémy, D. (1995) Dynamic typing in polymorphic languages. *J. Funct. Program.* **5**(1): 111–130.

Cheney, J. and Hinze, R. (2002) Poor man's dynamics and generics. In: Chakravarty, M. M. (ed.), *Proceedings of the ACM SIGPLAN 2002 Haskell Workshop*. ACM Press.

Coquand, T. (1991) An algorithm for testing conversion in type theory. In: Huet, G. and Plotkin, G. (eds.), *Logical Frameworks* pp. 255–277. Cambridge University Press.

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

Crary, K., Weirich, S. and Morrisett, G. (2002) Intensional polymorphism in type erasure semantics. *J. Funct. Program.* **12**(6): 567–600.

Dubois, C., Rouaix, F. and Weis, P. (1995) Extensional polymorphism. *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 118–129.

Fegaras, L. and Sheard, T. (1996) Revisiting catamorphisms over datatypes with embedded functions (or, programs from outer space). *Conf. Record 23rd ACM SIGPLAN/SIGACT Symp. on Principles of Programming Languages, POPL'96, St. Petersburg Beach, FL, USA, 21–24 Jan. 1996*, pp. 284–294. ACM Press.

Girard, J.-Y. (1971) Une extension de l'interprétation de Gödel à l'analyse, et son application à l'élimination de coupures dans l'analyse et la théorie des types. In: Fenstad, J. E. (ed.), *Proceedings of the Second Scandinavian Logic Symposium* pp. 63–92. North-Holland Publishing Co.

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

Harper, R. and Morrisett, G. (1995) Compiling polymorphism using intensional type analysis. *Twenty-Second ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 130–141.

Hinze, R. (2000) Polytypic values possess polykinded types. In: Backhouse, R. and Oliveira, J. (eds), *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, pp. 2–27.

Hinze, R., Jeuring, J. and Löh, A. (2002) Type-indexed data types. In: Eerke Boiten, B. M. (ed.), *Proceedings of the Sixth International Conference on Mathematics of Program Construction (MPC 2002)*, pp. 148–174.

Jansson, P. and Jeuring, J. (1997) PolyP—A polytypic programming language extension. *Twenty-Fourth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pp. 470–482.

Jay, C. B., Bellè, G. and Moggi, E. (1998) Functorial ML. *J. Funct. Program.* **8**(6): 573–619.

https://doi.org/10.1017/S0956796806005879 Published online by Cambridge University Press

Löh, A., Clarke, D. and Juering, J. (2003) Dependency-style Generic Haskell. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. To appear.

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

Morrisett, G., Felleisen, M. and Harper, R. (1995) Abstract models of memory management. *FPCA95: Conference on Functional Programming Languages and Computer Architecture (FPLCA)*, pp. 66–77.

Odersky, M. and Läufer, K. (1996) Putting type annotations to work. *Conference Record of POPL '96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersberg Beach, Florida*. pp. 54–67.

Peyton Jones, S. (ed). (2003) *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press.

Pierce, B. C. and Turner, D. N. (1998) Local type inference. *Conference Record of POPL 98: The 25TH ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 252–265.

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

Ruehr, F. (1998) Structural polymorphism. In: Backhouse, R. and Sheard, T. (eds.), *Informal Proceedings Workshop on Generic Programming, WGP'98*, Marstrand, Sweden.

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

Schürmann, C., Despeyroux, J. and Pfenning, F. (2001) Primitive recursion for higher-order abstract syntax. *Theor. Comput. Sci.* **266**(1–2): 1–58.

Sheard, T. (1993) *Type parametric programming*. Tech. rept. CSE 93-018. Oregon Graduate Institute.

Stone, C. and Harper, R. (2000) Deciding type equivalence in a language with singleton kinds. *Twenty-Seventh ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* pp. 214–225.

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

Wadler, P. (1989) Theorems for free! *FPCA89: Conference on Functional Programming Languages and Computer Architecture (FPLCA)*.

Wadler, P. and Blott, S. (1989) How to make ad-hoc polymorphism less ad hoc. *Sixteenth ACMSIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)* pp. 60–76. ACM Press.

Washburn, G. and Weirich, S. (2003) Boxes go bananas: Encoding higher-order abstract syntax with parametric polymorphism. *ACM SIGPLAN International Conference on Functional Programming (ICFP)*, pp. 249–262.

Weirich, S. (2001) Encoding intensional type analysis. In: Sands, D. (ed.), *10th European Symposium on Programming (ESOP)*, pp. 92–106.

Weirich, S. (2002a) Higher-order intensional type analysis. In: Métayer, D. L. (ed), *11th European Symposium on Programming (ESOP)*, pp. 98–114.

Weirich, S. (2002b) *Programming With Types*. PhD thesis, Cornell University.