Higher-Order Intensional Type Analysis

Stephanie Weirich

Department of Computer Science, Cornell University
Ithaca, NY 14850, USA
sweirich@cs.cornell.edu

Abstract. Intensional type analysis provides the ability to analyze abstracted types at run time. In this paper, we extend that ability to higher-order and kind-polymorphic type constructors. The resulting language is elegant and expressive. We show through examples how it extends the repertoire of polytypic definitions and the domain of valid types for those definitions.

1 Polytypic Programming

Some functions are naturally defined by examining the type structure of their arguments. For example, a *polytypic* pretty printer can format any data structure by decomposing it into basic parts, guided by its argument's type. Without such analysis, one must write a separate pretty printer for every data type and constantly update each one as the data types evolve. Polytypic programming, on the other hand, 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 debuggers, comparison functions and mapping functions. The theory behind describing such operations has been developed in a variety of frameworks [1,2,4,8,12,14,17,18,27,28,30,31].

Nevertheless, no single existing framework encompasses all polytypic definitions. These systems are limited by what polytypic operations they may express and by what types they may examine. These deficiencies are unfortunate because advanced languages depend crucially on these features. Only some frameworks for polytypism may express operations over parameterized data structures, such as maps and folds [14,17,18,27]. Yet parametric polymorphism is essential to modern typed programming languages. It is intrinsic to functional programming languages, such as ML [21] and Haskell [24], and also extremely important to imperative languages such as Ada [16] and Java [3,11]. Furthermore, only some frameworks for polytypism may examine types with binding structure, such as polymorphic or existential types [2,4,30]. However, these types are becoming increasingly more important. Current implementations of the Haskell language [19,29] include a form of existential type and first class polymorphism. Existential types are particularly useful for implementing dynamically extensible systems that may be augmented at run time with new operations and new types of data [13]. Also, the extension of polytypic programming to an object-oriented language will require the ability to examine types with binding structure.

D. Le Métayer (Ed.): ESOP 2002, LNCS 2305, pp. 98–114, 2002.

What is necessary to accommodate all types and all operations? First, because a quantified type hides type information, the semantics of the language must provide that information at run time to examine polymorphic and existential types. Second, the class of polytypic operations including mapping functions, reductions, zipping functions and folds must be defined in terms of *higher-order type constructors* instead of types. Such type constructors are "functions" such as *list* or *tree*, that are parameterized by other types.[1]

There is no reason why one system should not be able to define polytypic operations over both type constructors and quantified types. In fact, the two abilities are complementary if we represent quantified types with type constructors, using higher-order abstract syntax [25,30]. For example, we may represent the type $\forall\alpha.\alpha \to \alpha$ as the constant $\forall_\star$ applied to the type function $(\lambda\alpha{:}\star.\alpha \to \alpha)$.

In this paper, we address the previous limitations of polytypic programming and demonstrate how well these abilities fit together by extending Harper and Morrisett's seminal type-passing framework of *intensional type analysis* [12] to higher-order polytypism. In their language λ_i^{ML}, polytypic operations are defined by run-time examination of the structure of first-order types with the special term *typerec*. In λ_i^{ML}, an analyzable type is either *int*, *string*, a product type composed of two other types, or a function type composed of two other types. As these simple type constructors form an inductive datatype, *typerec* defines a fold (or catamorphism) over its type argument. For example, the result of analyzing types such as $\tau_1 \times \tau_2$ is defined in terms of analyses of τ_1 and τ_2. With the inclusion of type constructors that take a higher-order argument (such as $\forall_\star$ with argument of kind $\star \to \star$) the type structure of the language is no longer inductive. Previously, Trifonov *et al.* [30] avoided this issue by using the kind-polymorphic type constructor $\forall$ of kind $\forall\chi.(\chi \to \star) \to \star$ instead of $\forall_\star$ to represent kind-polymorphic types. As the argument of $\forall$ does not have a negative occurrence of the kind $\star$, the type structure remains inductive.

Hinze [14] has observed that we may define polytypic operations over type constructors by viewing a polytypic definition as an *interpretation* of the entire type constructor language, instead of a fold over the portion of kind type. However, his framework is based on compile-time definitions of polytypic functions (as opposed to run-time type analysis) and so cannot instantiate these functions with polymorphic or existential types. Here, we use this idea to extend Harper and Morrisett's *typerec* to a run-time interpreter for the type language, and extend it to higher-order type constructors and quantified types.

In the rest of this section, we review λ_i^{ML} and Hinze's framework for polytypic programming. In Section 2 we extend *typerec* to constructors of function kind. Because a polytypic definition is a model of the type language, it inhabits a unary *logical relation* indexed by the kind of the argument type constructor. A simple generalization in Section 3 extends this *typerec* to inhabit multi-place logical relations. Furthermore, in Section 4 we generalize *typerec* to constructors

[1] Just as terms are described by types, type constructors are described by kinds κ. The kind $\star$ contains all types. Higher-order constructors (functions from kind κ_1 to kind κ_2) have kind $\kappa_1 \to \kappa_2$.

of polymorphic kind. This extension admits the analysis of the $\forall$ constructor and encompasses as a special case the previous approach of Trifonov *et al.* [30]. Also, incorporating kind polymorphism enables further code sharing; without it, polytypic definitions must be duplicated for each kind of type argument. Finally, in Sections 5 and 6 we compare our approach with other systems and conclude with ideas for future extension.

1.1 Intensional Type Analysis

Harper and Morrisett's language λ_i^{ML} [12] introduced intensional type analysis with the *typerec* term. For example, *typetostring* (of type $\forall\alpha{:}\star\,.\,string$) uses *typerec* to produce a string representation of any type.[2]

$$
\begin{array}{l}
typetostring = \Lambda\alpha{:}\star\,.\ typerec[\lambda\beta{:}\star\,.\,string]\ \alpha \\
\quad int \quad \Rightarrow \texttt{"int"} \\
\quad string \Rightarrow \texttt{"string"} \\
\quad \rightarrow \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda x{:}\,string\,.\Lambda\gamma{:}\star\,.\lambda y{:}\,string\,.\texttt{"("} +\!\!+\ x\ +\!\!+\ \texttt{" -> "} +\!\!+\ y\ +\!\!+\ \texttt{")"} \\
\quad \times \quad \Rightarrow \Lambda\beta{:}\star\,.\lambda x{:}\,string\,.\Lambda\gamma{:}\star\,.\lambda y{:}\,string\,.\texttt{"("} +\!\!+\ x\ +\!\!+\ \texttt{" * "} +\!\!+\ y\ +\!\!+\ \texttt{")"}
\end{array}
$$

The annotation $[\lambda\beta{:}\star\,.\,string]$ on *typerec* above is used for type-checking. If α, the argument to *typerec*, is instantiated with the type *int* or *string*, this term immediately returns the appropriate string. If $\alpha$is a product or function type (in the $\times$ and $\rightarrow$ branches), *typerec* inductively calls itself to provide the strings of the subcomponents of the type. In these branches, the type variables β and γ are bound to the subcomponent types, and the term variables x and y are bound to the inductively computed strings. The rules below show the operation of *typerec* over various arguments, providing the inductively computed results to the product and function branches.

$$
\begin{array}{l}
typerec[c]\ \ int\ \ \overline{e} \mapsto e_{int} \\
typerec[c]\ \ string\ \ \overline{e} \mapsto e_{string} \\
typerec[c]\ (c_1 \times c_2)\ \overline{e} \mapsto e_\times\ [c_1]\ (typerec[c]\ c_1\ \overline{e})\ [c_2]\ (typerec[c]\ c_2\ \overline{e}) \\
typerec[c]\ (c_1 \rightarrow c_2)\ \overline{e} \mapsto e_\rightarrow\ [c_1]\ (typerec[c]\ c_1\ \overline{e})\ [c_2]\ (typerec[c]\ c_2\ \overline{e})
\end{array}
$$

The symbol $\overline{e}$ abbreviates the branches of the *typerec* ($int \Rightarrow e_{int}, string \Rightarrow e_{string}, \rightarrow \Rightarrow e_\rightarrow, \times \Rightarrow e_\times$). In this paper we will be deliberately vague about what type constructors comprise these branches and add new branches as necessary.

What is the return type of a *typerec* term? The typing judgment below reflects that this term is an induction over the structure of the analyzed type, c'. The annotation c applied to the argument type c' forms the return type of the *typerec* expression. For example, in *typetostring* above, the result type is $(\lambda\beta{:}\star.\,string)\alpha$ or *string*. In each branch, c' is specialized. For example, the *int* branch is of type $c\ int$, while the product branch takes two arguments of type $c\ \alpha$ and $c\ \beta$ to an expression of type $c(\alpha \times \beta)$. The context Γ contains assumptions about the types and kinds of the free type and term variables found inside c, c' and $\overline{e}$.

$$
\frac{\begin{array}{c}
\Gamma \vdash c' : \star \qquad \Gamma \vdash c : \star \rightarrow \star \qquad \Gamma \vdash e_{int} : c\ int \quad \Gamma \vdash e_{string} : c\ string \\
\Gamma \vdash e_\rightarrow : \forall\alpha{:}\star\,.c\ \alpha \rightarrow \forall\beta{:}\star\,.c\ \beta \rightarrow c(\alpha \rightarrow \beta) \\
\Gamma \vdash e_\times : \forall\alpha{:}\star\,.c\ \alpha \rightarrow \forall\beta{:}\star\,.c\ \beta \rightarrow c(\alpha \times \beta)
\end{array}}{\Gamma \vdash typerec[c]\ c'\ \overline{e} : c\ c'}
$$

[2] We use $+\!\!+$ as an infix function for string concatenation.

$$
\begin{array}{l}
size\langle\alpha\rangle\eta = \eta(\alpha) \\
size\langle\lambda\alpha{:}\kappa.c\rangle\eta = \Lambda\alpha{:}\kappa.\lambda x{:}Size\langle\kappa\rangle\alpha.(size\langle c\rangle\eta\{\alpha \mapsto x\}) \\
size\langle c_1 c_2\rangle\eta = (size\langle c_1\rangle\eta)\ [c_2]\ (size\langle c_2\rangle\eta) \\
size\langle int\rangle\eta = \lambda x{:}\, int\,.0 \\
size\langle string\rangle\eta = \lambda x{:}\, string\,.0 \\
size\langle\times\rangle\eta = \Lambda\alpha{:}\star.\lambda x{:}(\alpha \to int).\Lambda\beta{:}\star.\lambda y{:}(\beta \to int).\lambda v : \alpha \times \beta.x(\pi_1 v) + y(\pi_2 v) \\
size\langle+\rangle\eta = \Lambda\alpha{:}\star.\lambda x{:}(\alpha \to int).\Lambda\beta{:}\star.\lambda y{:}(\beta \to int). \\
\qquad\qquad \lambda v : \alpha + \beta.\ case\ v(inj_1\ w \Rightarrow xw \mid inj_2\ w \Rightarrow yw)
\end{array}
$$

Fig. 1. *size*

However, *typerec* may not express all polytypic definitions. For example, we cannot use it to define a term of type $\forall\alpha{:}\star \to \star.\forall\beta{:}\star.\alpha\ \beta \to int$, that counts the number of values of type β in a data structure of type $\alpha\ \beta$. Call this operation *fsize*. For example, if $c_1 = \lambda\alpha{:}\star.\alpha \times int$ and $c_2 = \lambda\alpha{:}\star.\alpha \times \alpha$, then $fsize[c_1]$ and $fsize[c_2]$ are constant functions returning 1 and 2 respectively. If α is instantiated with *list*, $fsize[list]$ is the standard length function.

As α is of higher-kind, we must apply it to some type in order to analyze it. We might try to define *fsize* as

$$fsize = \Lambda\alpha{:}\star \to \star.\Lambda\beta{:}\star\,.\ typerec[\lambda\gamma.\gamma \to int](\alpha\ \beta)\ldots$$

However, this approach is not correct. At run time, β will be instantiated before *typerec* analyzes $(\alpha\ \beta)$. The value returned by *typerec* will depend on what β is replaced with. If β is instantiated by *int*, then $c_1\beta$ and $c_2\beta$ will be the same type, and analysis cannot produce different results. Therefore, to define *fsize* we must analyze α independently of β.

1.2 Higher-Order Polytypism

How should we extend *typerec* to higher-order type constructors? What should the return type of such an analysis be? Hinze [14] observed that a polytypic definition should be an *interpretation* of the type language with elements of the term language. This interpretation must sound — *i.e.* when two types are equal, their interpretations are equal — so that we can reason about the behavior of a polytypic definition. A sound interpretation of higher-order types is to interpret type functions as term functions and type application as term application. Then β-equality between types $(i.e.(\lambda\alpha{:}\kappa.c_1)c_2 = c_2\{c_1/\alpha\})$ will be preserved by β-equality in the term language. The constants of the type language $(int, string, \to, \times)$ may be mapped to any term (of an appropriate type) providing the flexibility to define a number of different polytypic operations.

For example, the definition of the polytypic operation *size* is in Figure 1. This operation is defined by induction over a type constructors c. It is also parameterized by a finite map η (an environment) mapping type variables to terms. We use $\emptyset$ as the empty map, extend a map with a new mapping from the type variable α to the term e with the notation $\eta\{\alpha \mapsto e\}$, and retrieve the

mapping for a type variable with $\eta(\alpha)$. All variables in the argument of *size* should be in the domain of η. The first three lines of the definition in this figure are common to polytypic definitions. The definition for variables is determined by retrieving the mapping of the variable from environment. The environment is extended in the definition of *size* for type functions ($\lambda\alpha{:}\kappa.c$). As a type function is of higher kind, it is defined to be a polymorphic function from the *size* of the type argument, to the *size* of the body of the type constructor, with the environment updated to provide a mapping for the type variable occurring in the body. The type of x is determined by the kind of $\alpha$and is explained in the following. Because a type function maps to a polymorphic term function, a type application produces a term application.

The last four cases determine the behavior of *size*. Intuitively, *size* produces an iterator over a data structure that adds the "sizes" of all of its parts. We would like to use this operation in the definition of *fsize* as follows. Because *list* is a type constructor, the specialization $size\langle list\rangle$ maps a function to compute the "size" of values of some type β, to a function to compute the "size" of the entire list of type *list* β. If we supply the constant function $\lambda x{:}\beta.1$ for the list elements, we produce the desired length function for lists. Therefore, we may define *fsize* specialized by any closed type constructor c as $\Lambda\beta{:}\star.size\langle c\rangle[\beta](\lambda x{:}\beta.1)$.[3] For base types, such as *int* or *string*, *size* produces the constant function $\lambda x.0$, because they should not be included in computing the size. The type constructors $+$ and $\times$ are both parameterized by the two subcomponents of the $+$ or $\times$ types (α and β) and functions to compute their sizes (x and y).

For example, we can use many of the above definitions to compute $size\langle\lambda\alpha.\alpha\times string\rangle$. The slightly simplified result, when all of the definitions have been applied, is below. It is a function that when given an argument to compute the size of terms of type α, should accept a pair and apply this argument to the first component of the pair. (As the second component of the pair is of type *string*, its *size* is 0).

$$size\langle\lambda\alpha.\alpha\times string\rangle = \Lambda\alpha{:}\star.\lambda w{:}(\alpha\to int).\lambda v{:}(\alpha\times string).w(\pi_1 v)+0$$

Because type functions are mapped to term functions, the *type* of the polytypic definition (such as *size*) will be determined by the *kind* of the type constructor analyzed. In each instance, the definition of $size\langle c\rangle$ will be of type $Size\langle\kappa\rangle c$ where κ is the kind of c and $Size\langle\kappa\rangle c$ is defined by induction on the structure of κ. If the constructor c is of kind $\star$, then $Size\langle\star\rangle c$, is a function type from c to *int*. Otherwise, if c is of higher kind then *size* is parameterized by a corresponding *size* argument for the type argument to c.

$$\begin{aligned} Size\langle\star\rangle c &= c\to int\\ Size\langle\kappa_1\to\kappa_2\rangle c &= \forall\alpha{:}\kappa_1.Size\langle\kappa_1\rangle\alpha\to Size\langle\kappa_2\rangle(c\alpha)\end{aligned}$$

Why does the definition of *size* make sense? Though *size* is determined by the syntax of a type, a type is actually an equivalence class of syntactic expressions. To be well-defined, a polytypic function must return equivalent terms for

[3] Unlike λ_i^{ML} where types are analyzed at run time, in this framework polytypic functions are created and specialized to their type arguments at compile-time, so we may not make $fsize\langle c\rangle$ polymorphic over c.

all equivalent types, no matter how the types are expressed. For example, $size$ instantiated with $(\lambda\alpha{:}\star.\alpha \times string)\ int$ must be equal to $size\ \langle int \times string\rangle$ because these two types are equal by β-equality. Because the term functions provide the necessary equational properties, the definition of $size$ is sound. Therefore, though the interpretations of the type operators $(int, \rightarrow, \times)$ may change for each polytypic operation, the interpretations of functions $(\lambda\alpha{:}\kappa.c)$, variables α, and applications $(c_1 c_2)$ remain constant in every polytypic definition. As a result, the types of polytypic operations can be expressed using the following notation.

Definition 1. *A* polykinded type*, written $c\langle\kappa\rangle c'$, where c is a type constructor of kind $\star \rightarrow \star$, and c' a type constructor of kind κ, is defined by induction on the structure of the kind κ by:*

$$c\langle\star\rangle c' = cc' \qquad c\langle\kappa_1 \rightarrow \kappa_2\rangle c' = \forall\alpha{:}\kappa_1.c\langle\kappa_1\rangle\alpha \rightarrow c\langle\kappa_2\rangle(c'\alpha)$$

For example, we express $Size\langle\kappa\rangle c$ in this notation as $(\lambda\alpha{:}\star.\alpha \rightarrow int)\langle\kappa\rangle c$.

2 The Semantics of Higher-Order *typerec*

Hinze's framework specifies how to define a polytypic function at compile time by translating closed types into terms. However, in some cases, such as in the presence of polymorphic recursion, first-class polymorphism, or separate compilation it is not possible to specialize all type abstractions at compile time. Therefore, we extend a language supporting run-time type analysis to polytypic definitions over higher-order type constructors. We do so by changing the behavior of λ_i^{ML}'s *typerec* to be an *interpreter* of the type language at run time.

There is a close correspondence between the polykinded types and the typing judgment for *typerec*. Each of the branches of *typerec* may be written as a polykinded type. For example, the branch $e_\times$ is of type $c\langle\star \rightarrow \star \rightarrow \star\rangle\times = \forall\alpha{:}\star.c\alpha \rightarrow \forall\beta{:}\star.c\beta \rightarrow c(\alpha \times \beta)$. Carrying the analogy further suggests that we may extend *typerec* to all type constructors by relaxing the restriction that the argument to *typerec* be of kind $\star$, and by using a polykinded type to describe the result of *typerec*. We use $\oplus$ to notate arbitrary type constructor constants (such as $int, \rightarrow, \times$, called operators) and assume each $\oplus$ is of kind $\kappa_\oplus$.

$$\frac{\Gamma \vdash c' : \kappa \qquad \Gamma \vdash c : \star \rightarrow \star \qquad \Gamma \vdash e_\oplus : c\langle\kappa_\oplus\rangle\oplus \qquad (\forall e_\oplus \in \overline{e})}{\Gamma \vdash typerec[c]\, c'\overline{e} : c\langle\kappa\rangle c'}$$

Unfortunately, this judgment is not complete. As in the definition of $size\langle c\rangle\eta$, the operational semantics for higher-order *typerec* must involve some sort of environment η and the typing judgment must describe that environment.

In the following, we introduce higher-order *typerec* and describe how to typecheck a *typerec* term. We conclude this section with a number of examples demonstrating *typerec* extended to type constructors with binding constructs. To make these examples concrete, we change the semantics of the *typerec* term of λ_i^{ML}. The syntax of this language appears in Figure 2; we refer the reader to other sources [12,23] for the semantics not involved with *typerec*. Type constructors

$$
\begin{array}{ll}
(\mathit{kinds})\ \kappa ::= \star \mid \kappa_1 \to \kappa_2 & (\mathit{op's})\ \ \oplus ::= \mathit{int} \mid \to \mid \times \mid + \mid \ldots \\
(\mathit{con's})\ c ::= \alpha \mid \lambda\alpha{:}\kappa.c \mid c_1 c_2 \mid \oplus & (\mathit{types})\ \sigma ::= T(c) \mid \mathit{int} \mid \sigma \to \sigma \mid \forall\alpha{:}\kappa.\sigma \mid \ldots \\
(\mathit{exp's})\ e ::= i \mid x \mid \lambda x{:}\sigma.e \mid e_1 e_2 \mid \mathit{fix}\ x{:}\sigma.e \mid \Lambda\alpha{:}\kappa.e \mid e[c] \mid \mathit{typerec}[c][\Gamma,\eta,\rho]\ c\ \ \overline{e}\ \mid \ldots
\end{array}
$$

Fig. 2. Syntax

and types are separate syntactic classes in this language, with an injection $T(c)$ between the type constructors of kind $\star$ and the types. Consequently, we must slightly modify the definition of the base case of a polykinded type so that it produces a type instead of a type constructor: $c\langle\star\rangle c' = T(cc')$.

We define the operational semantics for higher-order *typerec* by structural induction on its type constructor argument.

$$
\begin{array}{l}
\mathit{typerec}[c][\Gamma',\eta,\rho]\ \alpha\ \ \overline{e} \mapsto \eta(\alpha) \\
\mathit{typerec}[c][\Gamma',\eta,\rho]\ (c_1 c_2)\ \ \overline{e} \mapsto (\mathit{typerec}[c][\Gamma',\eta,\rho]\ c_1\ \ \overline{e})[\rho(c_2)](\mathit{typerec}[c][\Gamma',\eta,\rho]\ c_2\ \ \overline{e}) \\
\mathit{typerec}[c][\Gamma',\eta,\rho]\ (\lambda\alpha{:}\kappa.c)\ \ \overline{e} \mapsto \\
\quad \Lambda\beta{:}\kappa.\lambda x{:}c\langle\kappa\rangle\beta.\ \mathit{typerec}[c][\Gamma'\{\alpha \mapsto \kappa\}, \eta\{\alpha \mapsto x\}, \rho\{\alpha \mapsto \beta\}]\ c\ \ \overline{e} \\
\mathit{typerec}[c][\Gamma',\eta,\rho]\ \ \oplus\ \ \overline{e} \mapsto e_\oplus
\end{array}
$$

The environment component η of *typerec* interprets the free type variables in its argument. For type checking (see below) the context Γ' lists the kinds of these variables. When analysis reaches a variable, *typerec* uses η to provide the appropriate value. For the analysis of type application $c_1 c_2$, *typerec* applies the analyzed constructor function c_1 to the analyzed argument c_2. In this rule, we must be careful that the free type variables in c_2 do not escape their scope, so we replace all of the free type variables occurring in c_2. For this substitution, we add an additional environment ρ mapping type variables to types. We substitute of all free variables of c_2 in the domain of ρ with $\rho(c_2)$. When the argument to *typerec* is a type constructor abstraction, the context and the term and type environments are extended. For operators, *typerec* returns the appropriate branch.

A reassuring property of this *typerec* is that it derives the original operational rules. For example, λ_i^{ML}'s *typerec* has the following evaluation for product types:

$$\mathit{typerec}[c]\ (c_1 \times c_2)\ \ \overline{e} \mapsto e_\times[c_1]\ (\mathit{typerec}[c]\ c_1\ \ \overline{e})\ [c_2]\ (\mathit{typerec}[c]\ c_2\ \ \overline{e})$$

With higher-order type analysis, because $c_1 \times c_2$ is the operator $\times$ applied to c_1 and c_2, the rule for type-constructor application generates the same behavior.

To typecheck a *typerec* term the context Γ' below describes the kinds of the variables in the domain of η and ρ. To check that Γ', η and ρ are well-formed, we formulate a new judgment $\Gamma; c \vdash \Gamma' \mid \eta \mid \rho$. This judgment is derived from two inference rules. The first rule states that the empty context and the empty environments are always valid.

$$\frac{}{\Gamma; c \vdash \emptyset \mid \emptyset \mid \emptyset} \qquad \frac{\Gamma; c \vdash \Gamma' \mid \eta \mid \rho \quad \Gamma \vdash c' : \kappa \quad \Gamma \vdash e : c\langle\kappa\rangle c' \quad \alpha \notin \mathit{Dom}(\Gamma, \Gamma')}{\Gamma; c \vdash \Gamma'\{\alpha \mapsto \kappa\} \mid \eta\{\alpha \mapsto e\} \mid \rho\{\alpha \mapsto c'\}}$$

In the second rule, if we add a new type variable α of kind κ to Γ', its mapping in ρ must be to a type constructor c' also of kind κ, and its mapping

in η must be to a term with type indexed by κ. Note that as we add to Γ' only type variables that are not in Γ, the domains of Γ and Γ' must be disjoint. With this judgment, we can state the formation rule for *typerec*.

$$\frac{\Gamma \vdash c : \star \to \star \quad \Gamma; c \vdash \Gamma' \mid \eta \mid \rho \quad \Gamma, \Gamma' \vdash c' : \kappa' \quad \Gamma \vdash e_\oplus : c\langle \kappa_\oplus \rangle \oplus \qquad (\forall e_\oplus \in \overline{e})}{\Gamma \vdash \mathit{typerec}[c][\Gamma', \eta, \rho]\, c'\ \overline{e} : c\langle \kappa' \rangle (\rho(c'))}$$

This rule and the rules for the dynamic semantics are appropriate because they satisfy type preservation. Looking at the four operational rules for *typerec*, we can see that no matter which one applies, if the original term was well-typed then the resulting term also has the same type. Furthermore, a closed, well-typed *typerec* term is never stuck; for any type constructor argument, one of the four operational rules must apply. These two properties may be used to syntactically prove type safety for this language [32].

We may implement *size* with higher-order *typerec* below (when Γ', η and ρ are empty, we elide them):

$$\begin{array}{lll} \mathit{size} = \Lambda\alpha{:}\star \to \star.\ \mathit{typerec}[\lambda\beta{:}\star.\beta \to \mathit{int}]\ \alpha \\ \quad \mathit{int} & \Rightarrow \lambda y{:}\mathit{int}.0 \\ \quad \mathit{string} & \Rightarrow \lambda y{:}\mathit{string}.0 \\ \quad \times & \Rightarrow \Lambda\beta{:}\star.\lambda x{:}\beta \to \mathit{int}.\Lambda\gamma{:}\star.\lambda y{:}\gamma \to \mathit{int}.\lambda v{:}\beta \times \gamma.\ x(\pi_1 v) + y(\pi_2 v) \\ \quad + & \Rightarrow \Lambda\beta{:}\star.\lambda x{:}\beta \to \mathit{int}.\Lambda\gamma{:}\star.\lambda y{:}\gamma \to \mathit{int}. \\ & \qquad \lambda v{:}\beta + \gamma.\ \ \mathit{case}\ \ v\ \ (\mathit{inj}_1\, z \Rightarrow x(z) \mid \mathit{inj}_2\, z \Rightarrow y(z)) \end{array}$$

This example demonstrates a few deficiencies of the calculus presented so far. First, what about recursive types? We cannot compute *size* for lists and trees without them. What about polymorphic or existential types? Must we limit *size* to constructors of kind $\star \to \star$, even though *typerec* can operate over constructors of any kind? We address these limitations in the rest of the paper.

2.1 Recursive Types

We have two choices to add recursive types to our system. Both versions are created with the type constructor $\mu_\star$ (of kind $(\star \to \star) \to \star$). In the first case, an *equi-recursive* type is definitionally equivalent to its unrolling, *i.e.* $\mu_\star c = c(\mu_\star c)$. Therefore, we must make analysis of $(\mu_\star c)$ equal to that of $(c(\mu_\star c))$. We do so with an evaluation rule for *typerec* that takes the fixed point of its argument as the interpretation of a recursive type[4]

$$\mathit{typerec}[c][\Gamma, \eta, \rho]\ \mu_\star\ \overline{e} \ \mapsto \Lambda\alpha{:}\star \to \star.\lambda x{:}(c\langle \star \to \star \rangle \alpha).\, \mathit{fix}\, f{:}(c\langle \star \rangle \mu_\star \alpha).\ (x[\mu_\star \alpha] f)$$

The alternative is to include *iso-recursive* types: those that require explicit term coercions. In other words, there is no equational rule for $\mu_\star$, but the calculus includes two terms that witness the isomorphism.

$$\mathit{roll}_{\mu_\star c} : c(\mu_\star c) \to \mu_\star c \qquad \mathit{unroll} : \mu_\star c \to c(\mu_\star c)$$

[4] In a call-by-value calculus this rule is ill-typed because we are taking the fixed point of an expression that is not necessarily of function type. To support this rule in such a calculus we would require that c return a function type for any argument.

With iso-recursive types, we have the most flexibility in the definition of polytypic functions. Without an equivalence rule governing $\mu_\star$, we are free to interpret it in any manner, as long as its branch in *typerec* has the correct type determined by the kind of $\mu_\star$. For a given c, this type is

$$c\langle(\star \to \star) \to \star\rangle\mu_\star = \forall\alpha{:}\star \to \star.[\forall\beta{:}\star\,.T(c\beta) \to T(c(\alpha\beta))] \to T(c(\mu_\star\alpha))$$

In most polytypic terms, the *typerec* branch for iso-recursive $\mu_\star$ will match the evaluation rule for equi-recursive $\mu_\star$.[5] For example, the $\mu_\star$ branch for *size* is below. The difference between it and the rule for equi-recursive types is an η-expansion around $x[\mu_\star\alpha]f$ that allows the explicit *unroll* coercion.

$$\begin{array}{c}\mu_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda x{:}(\forall\beta{:}\star\,.T(\beta \to \mathit{int}) \to T(\alpha\beta \to \mathit{int})).\\ \quad \mathit{fix}\, f{:}T(\mu_\star\alpha \to \mathit{int}).\lambda y{:}T(\mu_\star\alpha).\ x\ [\mu_\star\alpha]\ f\ (\mathit{unroll}\, y)\end{array}$$

In this branch, α is the body of the recursive type, and x is the result of *typerec* over that body. The definition of *size* for a recursive type should be a recursive function that accepts an argument y of recursive type, unrolls it to type $T(\alpha(\mu_\star\alpha))$, and calls x to produce *size* for this object. The call to x needs an argument that computes the size of $\mu_\star\alpha$. This argument is the result we are computing in the first place. Therefore, we use *fix* to name this result f and supply it to x.

2.2 F2 Polymorphism

The type constructor constants $\forall_\star$ and $\exists_\star$ (of kind $(\star \to \star) \to \star$) use higher-order abstract syntax [25] to describe polymorphic and existential[6] types of F2 [10,26,22]. These types are a subset of the polymorphic and existential types of λ_i^{ML}— they may only abstract constructors of kind $\star$ instead of any kind. The relationship between these type constructors and the corresponding types are:

$$T(\forall_\star c) = \forall\alpha{:}\star\,.T(c\alpha) \qquad T(\exists_\star c) = \exists\alpha{:}\star\,.T(c\alpha)$$

We can extend *size* with a branch for $\exists_\star$. For this branch, we must provide a function to calculate the size of the hidden type. We use the constant function zero, as that is result of *size* for types.

$$\begin{array}{l}\exists_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}(\forall\beta{:}\star\,.T(\beta \to \mathit{int}) \to T(\alpha\beta \to \mathit{int})).\\ \quad \lambda x{:}T(\exists\alpha).\, \mathit{let}\langle\beta, y\rangle = \mathit{unpack}\, x\ \mathit{in}\ \ r\ [\beta]\ \ (\lambda x{:}\beta.0)\ \ y\end{array}$$

With *size* we were fortunate that we could compute the value of *size* for the hidden type of an existential without analyzing it, as it was a constant function. However, for many polytypic functions, the function we pass to operate on the hidden type may itself be polytypic. Often it is the polytypic function

[5] In Section 3 we discuss a example that does not.

[6] We create an object of existential type $(\exists\alpha{:}\kappa.\sigma)$ with the term $\mathit{pack}\langle c, e\rangle\ \mathit{as}\ \exists\alpha{:}\kappa.\sigma$ (where e has type $\sigma\{c/\alpha\}$) and destruct the existentially typed e_1 with the term $\mathit{let}\langle\beta, x\rangle = \mathit{unpack}\, e_1\ \mathit{in}\ e_2$ which binds β and x to the hidden type and term of e_1 within e_2.

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

$$\frac{\begin{array}{l}\Gamma; c \vdash \Gamma' \mid \eta \mid \rho_1 \mid \ldots \mid \rho_n \\ \Gamma, \Gamma' \vdash c' : \kappa \qquad \Gamma \vdash c : \star^n \to \star \\ \Gamma \vdash e_\oplus : c\langle\kappa_\oplus\rangle^n \oplus \ldots \oplus \qquad (\forall e_\oplus \in \overline{e})\end{array}}{\Gamma \vdash \mathit{typerec}^n[c][\Gamma', \eta, \rho_1 \ldots \rho_n]\ c'\ \ \overline{e} : c\langle\kappa\rangle^n \rho_1(c') \ldots \rho_n(c')}$$

$\boxed{e \mapsto e'}$

$$\begin{array}{c}\mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ \oplus\ \ \overline{e} \mapsto \eta_\oplus \\ \mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ \alpha\ \ \overline{e} \mapsto \eta(\alpha) \\ \mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ (c_1 c_2)\ \overline{e} \mapsto (\mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ c_1\ \overline{e}) \\ [\rho_1(c_2)] \ldots [\rho_n(c_2)]\ (\mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ c_2\ \overline{e}) \\ \mathit{typerec}^n[c][\Gamma', \eta, \rho_1, \ldots, \rho_n]\ (\lambda\alpha{:}\kappa.c')\ \ \overline{e} \mapsto \Lambda\beta_1{:}\kappa. \ldots \Lambda\beta_n{:}\kappa.\lambda x{:}c\langle\kappa\rangle^n \beta_1 \ldots \beta_n. \\ (\mathit{typerec}^n[c][\Gamma'\{\alpha \mapsto \kappa\}, \eta\{\alpha \mapsto x\}, \rho_1\{\alpha \mapsto \beta_1\}, \ldots, \rho_n\{\alpha \mapsto \beta_n\}]\ c'\ \ \overline{e})\end{array}$$

Fig. 3. Semantics for multi-place *typerec*

itself, called recursively. This fact is not surprising considering the impredicative nature of $\forall_\star$ and $\exists_\star$ types: since the quantifiers range over *all* types we need an appropriate definition at all types.

For example, consider the simple function *copy* that creates an identical version of its argument. At base types, it is an identity function, at higher types, it breaks apart its argument and calls itself recursively.

$$\begin{array}{l}\mathit{fix}\ \mathit{copy} : (\forall\alpha : \star. T(\alpha \to \alpha)). \\ \quad \Lambda\alpha : \star.\ \mathit{typerec}[\lambda\alpha{:}\star.\alpha \to \alpha]\ \alpha \\ \quad \mathit{int} \Rightarrow \lambda i{:}\mathit{int}.i \\ \quad \to \Rightarrow \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha \to \alpha).\Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta \to \beta).\lambda f{:}T(\alpha \to \beta).r_\beta \circ f \circ r_\alpha \\ \quad \times\ \Rightarrow \Lambda\alpha{:}\star.\lambda r_\alpha{:}T(\alpha \to \alpha).\Lambda\beta{:}\star.\lambda r_\beta{:}T(\beta \to \beta).\lambda x{:}T(\alpha \times \beta).\langle r_\alpha(\pi_1 x), r_\beta(\pi_2 x)\rangle \\ \quad \mu_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}\forall\beta{:}\star.T(\beta \to \beta) \to T(\alpha\beta \to \alpha\beta). \\ \qquad\quad \mathit{fix}\ f{:}T(\mu_\star\alpha \to \mu_\star\alpha).\lambda x{:}T(\mu_\star\alpha).\ \mathit{roll}\ (r\ [\mu_\star\alpha]\ f\ (\mathit{unroll}\ x)) \\ \quad \forall_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}\forall\beta{:}\star.T(\beta \to \beta) \to T(\alpha\beta \to \alpha\beta). \\ \qquad\quad \lambda x{:}T(\forall_\star\alpha).\Lambda\beta{:}\star.r\ (\mathit{copy}\ [\beta])\ (x[\beta]) \\ \quad \exists_\star \Rightarrow \Lambda\alpha{:}\star \to \star.\lambda r{:}\forall\beta{:}\star.T(\beta \to \beta) \to T(\alpha\beta \to \alpha\beta).\lambda x{:}T(\exists_\star\alpha). \\ \qquad\quad \mathit{let}\langle\beta, y\rangle = \mathit{unpack}\ x\ \mathit{in}\ \ \mathit{pack}\langle\beta, r\ \ (\mathit{copy}\ [\beta])\ \ y\rangle\ \mathit{as}\ \exists\beta{:}\star.\alpha\beta\end{array}$$

3 Multi-place Polykinded Types

Unfortunately, with the calculus we have just developed we cannot implement several important examples of polytypic programming. For example, consider generic map. Given a function f, this map copies a data-structure parameterized by the type α, replacing every component x of type α, with fx. For example, if map is specialized to lists, then its type is $\forall\alpha{:}\star.\forall\beta{:}\star.(\alpha \to \beta) \to (\mathit{list}\ a) \to (\mathit{list}\ \beta)$. However, while the operation of generic map is guided by the structure of the type constructor *list*, this type is not a polykinded type of the form $c\langle\star \to \star\rangle\mathit{list}$. By an analogy with logical relations, Hinze observed that by extending the definition of polykinded types in the following way, we may define generic map.

Definition 2. *A* multi-place polykinded type*, written* $c\langle\kappa\rangle^n c_1 \ldots c_n$*, where* c *is of kind* $\star_1 \to \ldots \to \star_n \to \star$ *and* c_i *is of kind* κ *for* $1 \leq i \leq n$*, is defined by induction on* κ *as:*

$$c\langle\star\rangle^n c_1 \ldots c_n = T(c\ c_1 \ldots c_n)$$
$$c\langle\kappa_1 \to \kappa_2\rangle^n c_1 \ldots c_n = \forall\beta_1{:}\kappa_1 \ldots \forall\beta_n{:}\kappa_1 . c\langle\kappa_1\rangle^n \beta_1 \ldots \beta_n \to c\langle\kappa_2\rangle^n (c_1\beta_1) \ldots (c_n\beta_n)$$

Now the type of generic map may be expressed as $\forall\alpha{:}\star \to \star.(\to)\langle\star \to \star\rangle^2 \alpha\ \alpha$. If map is instantiated with the type constructor *list*, we get the expected type:

$$(\to)\langle\star \to \star\rangle^2 \mathit{list}\ \mathit{list} = \forall\alpha{:}\star .\forall\beta{:}\star .(\alpha \to \beta) \to (\mathit{list}\ \alpha \to \mathit{list}\ \beta).$$

Generalizing the definition of polykinded types forces us to also generalize *typerec* to $\mathit{typerec}^n$ and expand ρ to a set of type environments $\rho_1 \ldots \rho_n$ (see Figure 3). On type abstraction, n type variables are abstracted and $\rho_1 \ldots \rho_n$ are extended with these variables. We use these environments to provide substitutions for the n type arguments in a type application. With $\mathit{typerec}^2$ we may implement map, essentially a two-place version of *copy*.

Surprisingly, we can write useful functions when n is zero, such as a version of *typetostring* below.[7] In this code, *gensym* creates a unique string for each variable name, and *let* $x = e_1$ *in* e_2 is the usual abbreviation for $(\lambda x{:}\sigma.e_2)e_1$.

```
typetostring : ∀α:⋆. string.
typetostring = Λα:⋆. typerec⁰[string] α
  int ⇒ "int"
  → ⇒ λx:string.λy:string. "(" x ++ " -> " ++ y ++ ")"
  × ⇒ λx:string.λy:string. "(" x ++ " * " ++ y ++ ")"
  μ⋆ ⇒ λr:string →string. let x = gensym () in "mu"++ x ++ "." ++ (r x)
  ∀⋆ ⇒ λr:string →string. let x = gensym () in "all"++ x ++ "." ++ (r x)
  ∃⋆ ⇒ λr:string →string. let x = gensym () in "ex"++ x ++ "." ++ (r x)
```

Note that this example does not follow the pattern of iso-recursive types, which would be $\mu_\star \Rightarrow \lambda r{:}\mathit{string} \to \mathit{string}.\mathit{fix}\, f{:}\mathit{string}.r f$. In that case, the string representation of a recursive type would be infinitely long, witnessing the fact that a recursive type is an infinitely large type.

4 Kind Polymorphism

Why is there a distinction between types σ, and type constructors c, necessitating the irritating conversion $T()$? The reason is that we cannot analyze all types. In particular, we cannot analyze polymorphic types where the kind of the bound variable is not $\star$. We may analyze only those types created with the constructor $\forall_\star$. Trifonov *et al.*[30] (hereafter TSS) use the term *fully-reflexive* to refer to a

[7] For comparison, we could have also extended the $\mathit{typerec}^1$ version of *typetostring* (in Section 1.1). In the new branches, r would be of type $\forall\alpha{:}\star.\mathit{string} \to \mathit{string}$ instead of $\mathit{string} \to \mathit{string}$ as above, so a dummy type argument must be supplied when r is used.

$$\kappa ::= \ldots \mid \chi \mid \forall\chi.\kappa \qquad \oplus ::= \ldots \mid \forall \mid \exists \mid \forall^+ \qquad c ::= \ldots \mid \Lambda\chi.c \mid c[\kappa] \mid c\langle\kappa\rangle^n c_1 \ldots c_n$$
$$\sigma ::= \ldots \mid \forall^+\chi.\sigma \qquad e ::= \ldots \mid \Lambda^+\chi.e \mid e[\kappa]^+$$

Fig. 4. Additions for kind polymorphism

calculus where analysis operations are applicable to all types, and argue that this property is important for a type analyzing language.

A naive idea to make this language *fully-reflexive* would be to limit polymorphism to that of F2, *i.e.*, allow types only of the form $\forall\alpha{:}\star.\sigma$. However, then we cannot express the type of the $e_{\forall_\star}$ branch as it quantifies over a constructor of kind $\star \to \star$. We could then extend the language to allow types that quantify over constructors of kind $\star \to \star$, and add a constructor $(\forall_{\star\to\star})$ of kind $((\star \to \star) \to \star) \to \star$, but then the $e_{\forall_{\star\to\star}}$ branch would quantify over variables of kind $(\star \to \star) \to \star$. In general, we have a vicious cycle: for each type that we add to the calculus, we need a more complicated type to describe its branch in *typerec*. We could break this cycle by adding an infinite number of type constructors $\forall_\kappa$, thereby allowing construction of all polymorphic types. However, then *typerec* would require an infinite number of branches to cover all such types.

TSS avoid having an infinite number of branches for polymorphic types by introducing *kind polymorphism*. By holding the kind of the bound variable abstract, they may write one branch for all such types. Furthermore, they require kind polymorphism to analyze polymorphic types. As their type analysis is based on structural induction, they cannot handle $\forall_\star$ with a negative occurrence of $\star$ in the kind of its argument. With kind polymorphism, the $\forall$ constructor has kind $\forall\chi.(\chi \to \star) \to \star$, without such a negative occurrence.

Our version of *typerec*, as it is not based on induction, can already analyze $\forall_\star$. So their second motivation for kind polymorphism does not apply. However, in this system with kind-indexed types, we do have a separate and additional reason for adding kind polymorphism – our higher-order *typerec* term is naturally kind polymorphic and we would like to express that fact in the type system.

Like TSS, we include two forms of kind polymorphism: First, we extend the type constructor language to F2 by adding kind variables (χ), polymorphic kinds ($\forall\chi.\kappa$), and type constructors supporting kind abstraction ($\Lambda\chi.c$) and application ($c[\kappa]$). This polymorphism allows us to express the kind of the $\forall$ and $\exists$ constructors. Second, we also allow terms to abstract ($\forall^+\chi.e$) and apply ($e[\kappa]^+$) kinds, so that the $\forall$ branch of *typerec* may be polymorphic over the domain kind. We use the constructor $\forall^+$ to describe the type of kind-polymorphic terms. This constructor is also represented with higher-order abstract syntax: it is of kind $(\forall\chi.\star) \to \star$, where its argument describes how the type depends on the abstract kind χ.

To extend type analysis to polymorphic kinds we must extend the definition of $c\langle\kappa\rangle\alpha$ for the new kind forms χ and $\forall\chi.\kappa$. Therefore, we add polykinded types to the type constructor language and the following axioms to judge equality of type constructors, including a new axiom for polymorphic kinds:

$$\Gamma \vdash c\langle\star\rangle^n c_1 \ldots c_n = cc_1 \ldots c_n : \star$$
$$\Gamma \vdash c\langle\kappa_1 \to \kappa_2\rangle^n c_1 \ldots c_n =$$
$$\quad \forall[\kappa_1](\lambda\alpha_1{:}\kappa_1 \ldots . \forall[\kappa_1](\lambda\alpha_n{:}k_1 . (c\langle\kappa_1\rangle^n \alpha_1 \ldots \alpha_n) \to c\langle\kappa_2\rangle^n (c_1\alpha_1) \ldots (c_n\alpha_n)) \ldots) : \star$$
$$\Gamma \vdash c\langle\forall\chi.\kappa\rangle^n c_1 \ldots c_n = \forall^+(\Lambda\chi.c\langle\kappa\rangle^n (c_1[\chi]) \ldots (c_n[\chi])) : \star$$

Furthermore, we must extend the operational semantics of *typerec* to cover arguments that are kind abstractions or kind applications. By the above definition, *typerec* must produce a kind polymorphic term when reaching a kind polymorphic constructor. Therefore, an argument to *typerec* of a polymorphic kind pushes the *typerec* through the kind abstraction. Likewise, when we reach a kind application during analysis, we propagate the analysis through.

$$\mathit{typerec}^n[c][\Gamma, \eta, \overline{\rho}]\ (\Lambda\chi.c)\ \ \overline{e} \mapsto \Lambda^+\chi.\ \mathit{typerec}^n[c][\Gamma, \eta, \overline{\rho}]\ (c[\chi])\ \ \overline{e}$$
$$\mathit{typerec}^n[c][\Gamma, \eta, \overline{\rho}]\ (c[\kappa])\ \ \overline{e} \mapsto (\mathit{typerec}^n[c][\Gamma, \eta, \overline{\rho}]\ c\ \ \overline{e})[\kappa]^+$$

With kind polymorphism, we express the type of *size* more precisely as $\forall^+\chi.\forall\alpha{:}\chi.T((\lambda\beta{:}\star.\beta \to \mathit{int})\langle\chi\rangle\alpha)$. We can also extend *size* to general existential types. Before, as $\exists_\star$ hides type constructors of kind $\star$, we used the constant zero function as the *size* of the hidden type. Here, because the hidden type constructor may be of any kind, we must use a recursive call to define *size*.

$$\exists \Rightarrow \Lambda^+\chi.\Lambda\alpha{:}\chi \to \star.\lambda r : \forall\beta{:}\chi.T(c\langle\chi\rangle\beta) \to T(\alpha\beta \to \mathit{int}).$$
$$\quad \lambda x{:}T(\exists[\chi]\alpha).\ \mathit{let}\ \langle\beta, y\rangle = \mathit{unpack}\ x\ \mathit{in}\ \ r\ [\beta]\ \ (\mathit{size}[\chi][\beta])\ \ y$$

4.1 Example: typetostring

Unfortunately, even though we may analyze the entire type language, we cannot extend *typetostring* to create strings of all constructors. As kind polymorphism is parametric, we cannot differentiate constructors with polymorphic kinds. However, by giving *typetostring* a kind-polymorphic type we can produce many string representations.

$$\mathit{typetostring} : \forall^+\chi.\forall\alpha{:}\chi.T(\mathit{string}\langle\chi\rangle^0)$$

How do we use *typetostring* to produce strings of higher-order type constructors? When χ is not $\star$, the result of *typetostring* is not a *string*. However, we may analyze $\mathit{string}\langle\chi\rangle^0$ to produce a string when χ is a function kind. Using a technique similar to *type-directed partial evaluation* [5] we may *reify* a term of type $\mathit{string}\langle\chi\rangle^0$ into a string. To do so, we require *app* and *lam* to create string abstractions and applications.

$$\mathit{lam} : (\mathit{string} \to \mathit{string}) \to \mathit{string}$$
$$\mathit{lam} = \lambda x{:}\,\mathit{string} \to \mathit{string}\ .\ \mathit{let}\ b = \mathit{gensym}()\ \mathit{in}$$
$$\quad \text{"(}\texttt{lambda}\text{" } ++b\ ++\text{"."}\ ++(xb)\ ++\text{")"}$$

$$\mathit{app} : \mathit{string} \to (\mathit{string} \to \mathit{string})$$
$$\mathit{app} = \lambda x{:}\,\mathit{string}\ .\lambda y{:}\,\mathit{string}\ .$$
$$\quad \text{"("}\ ++x\ ++\text{" "}\ ++y\ ++\text{")"}$$

Below, let $c = \lambda\alpha{:}\star.(\alpha \to \mathit{string}) \times (\mathit{string} \to \alpha)$

$$\mathit{ReifyReflect} = \mathit{typerec}[c]\ \alpha$$
$$\mathit{string} \Rightarrow \langle\lambda y{:}\,\mathit{string}\ .y, \lambda y{:}\,\mathit{string}\ .y\rangle$$
$$\to \quad \Rightarrow \Lambda\alpha_1{:}\star.\lambda r_1{:}c\alpha_1.\Lambda\alpha_2{:}\star.\lambda r_2{:}c\alpha_2.$$
$$\quad\quad \mathit{let}\langle \mathit{reify}_1, \mathit{reflect}_1\rangle = r_1; \langle \mathit{reify}_2, \mathit{reflect}_2\rangle = r_2\ \mathit{in}$$
$$\quad \langle\lambda y{:}\alpha_1 \to \alpha_2.\mathit{lam}(\mathit{reify}_2 \circ y \circ \mathit{reflect}_1), \lambda y{:}\,\mathit{string}\ .\mathit{reflect}_2 \circ \mathit{app}\ y\ \ \circ \mathit{reify}_1\rangle$$

The result of *reify*, the first component of *ReifyReflect* above, composed with *typetostring* is a string representation of the long $\beta\eta$-normal form of the type constructor. What if that constructor has a polymorphic kind? We cannot extend *ReifyReflect* to analyze $string\langle\forall\chi.\kappa\rangle^0$, because parametric kind polymorphism prevents us from writing the functions $klam : (\forall^+\chi.\, string) \rightarrow string$ and $kapp : string \rightarrow \forall\chi^+.\, string$.

We also need *ReifyReflect* to create string representations of polymorphic types. In the previous version of *typetostring*, for the constructor $\forall_\star$, the inductive argument r was of type $string \rightarrow string$. With kind polymorphism, the type of this argument $(T(string\langle\chi\rangle^0))$ is dependent on χ the kind abstracted by $\forall$. In order to call r, we need to manufacture a value of this type — we need to *reflect* a string into the appropriate argument for the inductive call in *typetostring*:

$$\begin{array}{l}\forall \;\Rightarrow\; \Lambda^+\chi.\;\; \Lambda\alpha{:}\chi \rightarrow \star.\;\; \lambda r{:}T(string\langle\chi\rangle^0) \;\rightarrow\; string\,. \\ \quad let\;\; \langle reify, reflect\rangle = ReifyReflect[string\langle\chi\rangle^0] \\ \qquad v = gensym\;\; ()\;\; in\;\; \texttt{"all"} \mathbin{+\!\!+} v \mathbin{+\!\!+} \texttt{"."} \mathbin{+\!\!+} (r\;\; (reflect\;\; v))\end{array}$$

Again, because *ReifyReflect* is limited to kinds of the form $\star$ or $\kappa_1 \rightarrow \kappa_2$, we can only accept the types of F_ω [10] (*i.e.*, types such as $\forall[\star \rightarrow \star](\lambda\alpha{:}\star \rightarrow \star.c)$ but not $\forall[\forall\chi.\kappa](\lambda\alpha{:}\forall\chi.\kappa.c)$). And just as we cannot extend *ReifyReflect* to kind-polymorphic constructors, we cannot extend *typetostring* to kind-polymorphic types (those formed by $\forall^+$). While this calculus is fully-reflexive, we cannot completely discriminate all of the type constructors of this language.

4.2 Analysis of Polymorphic Types

In Section 2, we were reassured when the operation of higher-order *typerec* over product types mirrored that of λ_i^{ML}. How does analysis of polymorphic and existential types differ when *typerec* is viewed as a structural induction (as in TSS) and as an interpretation of the type language?

In the first case (which we distinguish by $typerec^i$) we have the following operational rule for polymorphic types; when c' is analyzed, its argument β is also examined with the same analysis.

$$typerec^i[c]\;\; (\forall[\kappa]c')\;\; \overline{e} \;\mapsto\; e_\forall\;\; [\kappa]^+[c']\;\; (\Lambda\beta{:}\kappa.\; typerec^i[c]\;\; (c'\beta)\;\; \overline{e})$$

With higher-order *typerec*, we may derive the following rule for polymorphic types. Here, the result of analysis of the argument to c' may be supplied in x.

$$\begin{array}{l}typerec[c][\Gamma', \eta, \rho]\;\; (\forall[\kappa]c')\;\; \overline{e} \;\mapsto^* \\ \quad e_\forall\;\; [\kappa]^+[c']\;\; (\Lambda\beta{:}\kappa.\lambda x{:}T(c\langle\kappa\rangle\beta).\; typerec[c][\Gamma'\{\alpha \mapsto \kappa\}, \eta\{\alpha \mapsto x\}, \rho\{\alpha \mapsto \beta\}]\;\; (c'\alpha)\;\; \overline{e})\end{array}$$

However, many examples of polytypic functions defined by higher-order *typerec* (such as *copy*) create a fixed point of the Λ-abstracted *typerec* term, and it is this fixed point applied to β that eventually replaces x. Therefore, as above, the argument to c' is examined with the same analysis. The difference between the two versions is similar to the difference between iso- and equi-recursive types. Because we have more expressiveness in the analysis of type constructors with higher-order *typerec*, we have more flexibility in the analysis of quantified types. TSS's calculus may implement *copy* (though they must restrict the kind of its argument to $\star$) but not *typetostring*.

5 Related Work

In lifting type analysis to higher-order constructors, this work is related to induction over datatypes with embedded function spaces and, more specifically, to those datatypes representing higher-order abstract syntax. Meijer and Hutton [20] describe how to extend catamorphisms to datatypes with embedded functions by simultaneously computing an inverse. Fegaras and Sheard [9] simplify this process, noting that when the analyzed function is *parametric*, an inverse is not required. TSS employ their technique for the type level analysis of recursive types in the language λ_i^Q [30], using the kind language to enforce that the argument to $\mu_\star$ is parametric. Likewise, in a language for expressing induction over higher-order abstract syntax, Despeyroux *et al.* [6,7], use a modal type to indicate parametric functions. In this paper, because only terms analyze types, all analyzed type functions are parametric and so we do not require such additional typing machinery.

6 Conclusions and Future Work

The goal of this work is to extend polytypic programming to encompass the features of expressive and advanced type systems. Here, we provide an operational semantics for type constructor polytypism by extending *typerec* to cover higher-order types. By casting these operations in a type-passing framework, we extend polytypic definitions over type constructors (such as *size* and *map*) to situations where type abstraction cannot be specialized away at compile time. With type passing, we also extend the domain of polytypic definitions to include first-class polymorphic and existential types. With the addition of kind polymorphism and polykinded types, we allow the types of polytypic operations to be explicitly and accurately described. Finally, by extending *typerec* to constructors of polymorphic kind we allow the analysis of constructors such as $\forall$ and $\exists$ in a flexible manner and provide insight to the calculus of TSS.

We hope to extend this work in the future with type-level type analysis. The languages λ_i^{ML} and λ_i^Q include *Typerec* that analyzes types to produce other types. This operator is often important in describing the types of polytypic definitions. Hinze *et al.* [15] provide a number of examples of higher-order polytypic term definitions that require higher-order polytypic type definitions. However, adding an operator to analyze higher-order constructors will require machinery at the kind level like TSS's λ_i^Q [30] or Despeyroux *et al.* [6,7].

This work suggests other areas of future research. First, because this framework depends on a type-passing semantics, it is important to investigate and develop compiler optimizations that would eliminate unneeded run-time analysis. Furthermore, while intensional type analysis has traditionally been used in the context of type-based compilation, we would like to incorporate this system in an expressive user language. Finally, because this language supports the analysis of types with binding structure, it may be applicable to adding polytypic programming to object-oriented languages.

Acknowledgments. Thanks to Dan Grossman, Michael Hicks, Greg Morrisett, Steve Zdancewic and the anonymous reviewers for helpful suggestions.

References

1. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Gordon Plotkin. Dynamic typing in a statically-typed language. *ACM Transactions on Programming Languages and Systems*, 13(2):237–268, April 1991. 98
2. Martín Abadi, Luca Cardelli, Benjamin Pierce, and Didier Rémy. Dynamic typing in polymorphic languages. *Journal of Functional Programming*, 5(1):111–130, January 1995. 98, 98
3. Gilad Bracha, Martin Odersky, David Stoutamire, and Philip Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In Craig Chambers, editor, *Object Oriented Programming: Systems, Languages, and Applications (OOPSLA)*, pages 183–200, Vancouver, BC, 1998. 98
4. Karl Crary and Stephanie Weirich. Flexible type analysis. In *Proceedings of the Fourth ACM SIGPLAN International Conference on Functional Programming*, pages 233–248, Paris, September 1999. 98, 98
5. O. Danvy. Type-directed partial evaluation. In *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, St. Petersburg, Florida, January 1996*, pages 242–257, 1996. 110
6. Joëlle Despeyroux and Pierre Leleu. Recursion over objects of functional type. *Mathematical Structures in Computer Science*, 11:555–572, 2001. 112, 112
7. Joëlle Despeyroux, Frank Pfenning, and Carsten Schürmann. Primitive recursion for higher-order abstract syntax. In *Third International Conference on Typed Lambda Calculi and Applications*, volume 1210 of *Lecture Notes in Computer Science*, pages 147–163, Nancy, France, April 1997. Springer-Verlag. 112, 112
8. Catherine Dubois, François Rouaix, and Pierre Weis. Extensional polymorphism. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 118–129, San Francisco, January 1995. 98
9. L. Fegaras and T. Sheard. Revisiting catamorphisms over datatypes with embedded functions. In *Twenty-Third ACM Symposium on Principles of Programming Languages*, St. Petersburg Beach, Florida, January 1996. 112
10. Jean-Yves Girard. *Interprétation fonctionelle et élimination des coupures de l'arithmétique d'ordre supérieur.* PhD thesis, Université Paris VII, 1972. 106, 111
11. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification.* Addison-Wesley, 1996. 98
12. Robert Harper and Greg Morrisett. Compiling polymorphism using intensional type analysis. In *Twenty-Second ACM Symposium on Principles of Programming Languages*, pages 130–141, San Francisco, January 1995. 98, 99, 100, 103
13. Michael Hicks, Stephanie Weirich, and Karl Crary. Safe and flexible dynamic linking of native code. In R. Harper, editor, *Types in Compilation: Third International Workshop, TIC 2000; Montreal, Canada, September 21, 2000; Revised Selected Papers*, volume 2071 of *Lecture Notes in Computer Science*, pages 147–176. Springer, 2001. 98
14. Ralf Hinze. Polytypic values possess polykinded types. In Roland Backhouse and J.N. Oliveira, editors, *Proceedings of the Fifth International Conference on Mathematics of Program Construction (MPC 2000)*, Ponte de Lima, Portugal, July 2000. 98, 98, 99, 101

15. Ralf Hinze, Johan Jeuring, and Andres L oh. Type-indexed data types. Available at http://www.cs.uu.nl/~johanj/publications/publications.html, 2001. 112
16. International Organisation for Standardisation and International Electrotechnical Commission. *Ada Reference Manual: Language and Standard Libraries*, 6.0 edition, December 1994. International Standard ISO/IEC 8652:1995. 98
17. Patrick Jansson and Johan Jeuring. PolyP - a polytypic programming language extension. In *Twenty-Fourth ACM Symposium on Principles of Programming Languages*, pages 470–482, Paris, France, 1997. 98, 98
18. C.B. Jay, G. Bellè, and E. Moggi. Functorial ML. *Journal of Functional Programming*, 8(6):573–619, November 1998. 98, 98
19. Mark P Jones and Alastair Reid. *The Hugs 98 User Manual*. Yale Haskell Group and Oregon Graduate Institute of Science and Technology. Available at http://cvs.haskell.org/Hugs/pages/hugsman/index.html. 98
20. Erik Meijer and Graham Hutton. Bananas in space: Extending fold and unfold to exponential types. In *Conf. Record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 Int. Conf. on Functional Programming Languages and Computer Architecture, FPCA'95, La Jolla, San Diego, CA, USA, 25–28 June 1995*, pages 324–333. ACM Press, New York, 1995. 112
21. Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, Cambridge, Massachusetts, 1997. 98
22. John C. Mitchell and Gordon D. Plotkin. Abstract types have existential type. *ACM Transactions on Programming Languages and Systems*, 10(3):470–502, July 1988. 106
23. Greg Morrisett. *Compiling with Types*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, Pennsylvania, December 1995. 103
24. Simon L. Peyton Jones and J. Hughes (editors). Report on the programming language Haskell 98, a non-strict purely functional language. Technical Report YALEU/DCS/RR-1106, Yale University, Department of Computer Science, February 1999. Available from http://www.haskell.org/definition/. 98
25. Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 199–208, Atlanta, GA, USA, June 1988. 99, 106
26. John C. Reynolds. Types, abstraction and parametric polymorphism. In *Information Processing '83*, pages 513–523. North-Holland, 1983. Proceedings of the IFIP 9th World Computer Congress. 106
27. Fritz Ruehr. Structural polymorphism. In Roland Backhouse and Tim Sheard, editors, *Informal Proceedings Workshop on Generic Programming, WGP'98, Marstrand, Sweden, 18 June 1998.*, 1998. 98, 98
28. T. Sheard. Type parametric programming. Technical Report CSE 93-018, Oregon Graduate Institute, 1993. 98
29. The GHC Team. *The Glasgow Haskell Compiler User's Guide*, version 5.02 edition. Available at http://www.haskell.org/ghc/. 98
30. Valery Trifonov, Bratin Saha, and Zhong Shao. Fully reflexive intensional type analysis. In *Fifth ACM SIGPLAN International Conference on Functional Programming*, pages 82–93, Montreal, September 2000. 98, 98, 99, 99, 100, 108, 112, 112
31. Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 60–76. ACM, 1989. 98
32. Stephanie Weirich. *Programming With Types*. PhD thesis, Cornell University, 2002. Forthcoming. 105