Polytypic values possess polykinded types

December 1999

RALF HINZE
Institut für Informatik III, Universität Bonn
Römerstraße 164, 53117 Bonn, Germany
(*e-mail:* ralf@informatik.uni-bonn.de)

Abstract

A polytypic value is one that is defined by induction on the structure of types. In Haskell the type structure is described by the so-called kind system, which distinguishes between manifest types like the type of integers and functions on types like the list type constructor. Previous approaches to polytypic programming were restricted in that they only allowed to parameterize values by types of one fixed kind. In this paper we show how to define values that are indexed by types of arbitrary kinds. It appears that these polytypic values possess types that are indexed by kinds. We present several examples that demonstrate that the additional flexibility is useful in practice. One paradigmatic example is the mapping function, which describes the functorial action on arrows. A single polytypic definition yields mapping functions for datatypes of arbitrary kinds including first- and higher-order functors. Polytypic values enjoy polytypic properties. Using kind-indexed logical relations we prove among other things that the polytypic mapping function satisfies suitable generalizations of the functorial laws.

1 Introduction

It is widely accepted that type systems are indispensable for building large and reliable software systems. Types provide machine checkable documentation and are often helpful in finding programming errors at an early stage. Polymorphism complements type security by flexibility. Polymorphic type systems like the Hindley-Milner system (Milner, 1978) allow the definition of functions that behave uniformly over all types. However, even polymorphic type systems are sometimes less flexible that one would wish. For instance, it is not possible to define a polymorphic equality function that works for all types—the parametricity theorem (Wadler, 1989) implies that a function of type $\forall a.a \to a \to \mathit{Bool}$ must necessarily be constant. As a consequence, the programmer is forced to program a separate equality function for each type from scratch.

Polytypic programming (Bird *et al.*, 1996; Backhouse *et al.*, 1999) addresses this problem. Actually, equality serves as a standard example of a polytypic function that can be defined by induction on the structure of types. In a previous paper (Hinze, 2000) the author has shown that polytypic functions are uniquely defined by giving cases for primitive types, the unit type, sums, and products. Given this information a tailor-made equality function can be automatically generated for each user-defined type.

Another useful polytypic function is the so-called mapping function. The mapping function of a unary type constructor f applies a given function to each element of type a in a given structure of type $f\ a$—we tacitly assume that f does not include function types. Note that unlike equality the mapping function is indexed by a type constructor, that is, by a function on types. Now, mapping functions can be defined for type constructors of arbitrary arity. In the general case the mapping function takes n functions and applies the i-th function to each element of type a_i in a given structure of type $f\ a_1\ \ldots\ a_n$. Alas, current approaches to polytypic programming (Jansson & Jeuring, 1997; Hinze, 2000) do not allow to define these mapping functions at one stroke. The reason is simply that the mapping functions have different types for different arities.

This observation suggests a natural extension of polytypic programming: it should be possible to assign a type to a polytypic value that depends on the arity of the type-index. Actually, we are more ambitious in that we consider not only first-order but also higher-order type constructors. A type constructor is said to be higher-order if it operates on type constructors rather than on types. To distinguish between types, first-order and higher-order type constructors, they are often assigned so-called kinds (McCracken, 1979), which can be seen as the 'types of types'. Using the notion of kind we can state the central idea of this paper as follows: polytypic values possess types that are defined by induction on the structure of kinds. It turns out that the implementation of this idea is much simpler than one would expect.

The rest of this paper is organized as follows. Section 2 illustrates the approach using the example of mapping functions. Section 3 introduces the language of kinds and types, which is based on the simply typed lambda calculus. Section 4 explains how to define polytypic values and polykinded types and Section 5 shows how to specialize a polytypic value to concrete instances of datatypes. Section 6 presents several examples of polytypic functions with polykinded types, which demonstrate that the extension is useful in practice. Polytypic values enjoy polytypic properties. Section 7 shows how to express polytypic laws using kind-indexed logical relations. Among other things, we show that the polytypic mapping function satisfies suitable generalizations of the functorial laws. Finally, Section 8 reviews related work and Section 9 concludes.

2 A worked-out example: mapping functions

This section illustrates the central idea by means of a worked-out example: mapping functions. For concreteness, the code will be given in the functional programming language Haskell 98 (Peyton Jones & Hughes, 1999). Before tackling the polytypic mapping function let us first take a look at different datatypes and associated monotypic mapping functions. As an aside, note that the combination of a type constructor and its mapping function is referred to as a functor.

As first, rather simple examples consider lists and rose trees.

$$\begin{aligned}\mathbf{data}\ \mathit{List}\ a &= \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)\\ \mathbf{data}\ \mathit{Rose}\ a &= \mathit{Branch}\ a\ (\mathit{List}\ (\mathit{Rose}\ a))\end{aligned}$$

Both *List* and *Rose* are unary type constructors, that is, they map types to types. In Haskell 98 the 'type' of a type constructor is specified by the kind system. For instance, *List* and *Rose* have kind $\star \to \star$. The '$\star$' kind represents manifest types like *Int* or *Bool*. The kind $\kappa_1 \to \kappa_2$ represents type constructors that map type constructors of kind κ_1 to those of kind κ_2. The mapping functions for *List* and *Rose* are given by

$$\begin{aligned}&\mathit{mapList} &&:: \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{List}\ a_1 \to \mathit{List}\ a_2)\\ &\mathit{mapList}\ \mathit{mapa}\ \mathit{Nil} &&= \mathit{Nil}\\ &\mathit{mapList}\ \mathit{mapa}\ (\mathit{Cons}\ v\ \mathit{vs}) &&= \mathit{Cons}\ (\mathit{mapa}\ v)\ (\mathit{mapList}\ \mathit{mapa}\ \mathit{vs})\\ &\mathit{mapRose} &&:: \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{Rose}\ a_1 \to \mathit{Rose}\ a_2)\\ &\mathit{mapRose}\ \mathit{mapa}\ (\mathit{Branch}\ v\ \mathit{vs}) &&= \mathit{Branch}\ (\mathit{mapa}\ v)\\ & && \qquad\qquad (\mathit{mapList}\ (\mathit{mapRose}\ \mathit{mapa})\ \mathit{vs})\ .\end{aligned}$$

Both *mapList* and *mapRose* take a function of type $a_1 \to a_2$ and apply this function to elements of type a_1 within a list or a rose tree. It is perhaps unusual to call the argument function *mapa*. The reason for this choice will become clear as we go along. For the moment it suffices to note that the definitions of the mapping functions rigidly follow the structure of the corresponding datatype declarations.

Lists and rose trees are examples of so-called regular or uniform types. Briefly, a regular type is one that can be defined as the least fixpoint of a functor. Interestingly, Haskell's type system is expressive enough to rephrase *List* and *Rose* using an explicit fixpoint operator (Meijer & Hutton, 1995). We will repeat this construction in the following as it provides us with interesting examples of datatypes and associated mapping functions. First, we define the so-called base or pattern functors of *List* and *Rose*.

$$\begin{aligned}\mathbf{data}\ \mathit{ListF}\ a\ b &= \mathit{Nil} \mid \mathit{Cons}\ a\ b\\ \mathbf{data}\ \mathit{RoseF}\ a\ b &= \mathit{Branch}\ a\ (\mathit{List}\ b)\end{aligned}$$

The types *ListF* and *RoseF* have kinds $\star \to (\star \to \star)$, which shows that binary type constructors are curried in Haskell. The following definition introduces a fixpoint operator on the type level (**newtype** introduces a new type that is isomorphic to the type on the right-hand side).

$$\mathbf{newtype}\ \mathit{Fix}\ f \;=\; \mathit{In}\ (f\ (\mathit{Fix}\ f))$$

The kind of *Fix* is $(\star \to \star) \to \star$, that is, *Fix* has a second-order kind. The order of a kind is given by $\mathit{order}(\star) = 0$ and $\mathit{order}(\kappa_1 \to \kappa_2) = \mathit{max}\{1+\mathit{order}(\kappa_1), \mathit{order}(\kappa_2)\}$. It remains to define *List* and *Rose* as fixpoints of their base functors (**type** defines a type synonym).

$$\begin{aligned}\mathbf{type}\ \mathit{List}'\ a &= \mathit{Fix}\ (\mathit{ListF}\ a)\\ \mathbf{type}\ \mathit{Rose}'\ a &= \mathit{Fix}\ (\mathit{RoseF}\ a)\end{aligned}$$

Now, how can we define mapping functions for lists and rose trees thus defined? For a start, we define mapping functions for the base functors.

$$
\begin{array}{lcl}
\mathit{mapListF} & :: & \forall a_1\ a_2.(a_1 \to a_2) \to \forall b_1\ b_2.(b_1 \to b_2) \\
 & & \quad \to (\mathit{ListF}\ a_1\ b_1 \to \mathit{ListF}\ a_2\ b_2) \\
\mathit{mapListF}\ \mathit{mapa}\ \mathit{mapb}\ \mathit{Nil} & = & \mathit{Nil} \\
\mathit{mapListF}\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Cons}\ v\ w) & & \\
 & = & \mathit{Cons}\ (\mathit{mapa}\ v)\ (\mathit{mapb}\ w) \\
\mathit{mapRoseF} & :: & \forall a_1\ a_2.(a_1 \to a_2) \to \forall b_1\ b_2.(b_1 \to b_2) \\
 & & \quad \to (\mathit{RoseF}\ a_1\ b_1 \to \mathit{RoseF}\ a_2\ b_2) \\
\mathit{mapRoseF}\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Branch}\ v\ w) & & \\
 & = & \mathit{Branch}\ (\mathit{mapa}\ v)\ (\mathit{mapList}\ \mathit{mapb}\ w)
\end{array}
$$

Since the base functors have two type arguments, their mapping functions take two functions, *mapa* and *mapb*, and apply them to values of type a_1 and b_1, respectively. Even more interesting is the mapping function for *Fix*

$$
\begin{array}{lcl}
\mathit{mapFix} & :: & \forall f_1\ f_2.(\forall a_1\ a_2.(a_1 \to a_2) \to (f_1\ a_1 \to f_2\ a_2)) \\
 & & \quad \to (\mathit{Fix}\ f_1 \to \mathit{Fix}\ f_2) \\
\mathit{mapFix}\ \mathit{mapf}\ (\mathit{In}\ v) & = & \mathit{In}\ (\mathit{mapf}\ (\mathit{mapFix}\ \mathit{mapf})\ v)\ ,
\end{array}
$$

which takes a polymorphic function as argument. In other words, *mapFix* has a so-called rank-2 type signature (Leivant, 1983). As an aside, note that Haskell 98 does not support rank-2 types. Recent versions of GHC (Team, 1999) and Hugs (Jones & Peterson, 1999), however, provide the necessary extensions. The argument function, *mapf*, has a more general type than one would probably expect: it takes a function of type $a_1 \to a_2$ to a function of type $f_1\ a_1 \to f_2\ a_2$. By contrast, the mapping function for *List* (which also has kind $\star \to \star$) takes $a_1 \to a_2$ to $\mathit{List}\ a_1 \to \mathit{List}\ a_2$. The definitions below demonstrate that the extra generality is vital.

$$
\begin{array}{lcl}
\mathit{mapList}' & :: & \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{List}'\ a_1 \to \mathit{List}'\ a_2) \\
\mathit{mapList}'\ \mathit{mapa} & = & \mathit{mapFix}\ (\mathit{mapListF}\ \mathit{mapa}) \\
\mathit{mapRose}' & :: & \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{Rose}'\ a_1 \to \mathit{Rose}'\ a_2) \\
\mathit{mapRose}'\ \mathit{mapa} & = & \mathit{mapFix}\ (\mathit{mapRoseF}\ \mathit{mapa})
\end{array}
$$

In the definition of $\mathit{mapList}'$ the argument of *mapFix* has type $\forall b_1\ b_2.(b_1 \to b_2) \to (\mathit{ListF}\ a_1\ b_1 \to \mathit{ListF}\ a_2\ b_2)$, that is, f_1 is instantiated to $\mathit{ListF}\ a_1$ and f_2 to $\mathit{ListF}\ a_2$.

Remark 1

In the initial algebra semantics of datatypes the mapping function of the type functor List' is defined by

$$
\mathit{mapList}'\ \mathit{mapa} \quad = \quad (\!|\mathit{In} \cdot \mathit{mapListF}\ \mathit{mapa}\ \mathit{id}|\!)\ ,
$$

where $(\!|-|\!)$ is the so-called catamorphism of the list type constructor.

The list datatype is commonly used to represent sequences of elements. An alternative data structure, which supports logarithmic access, is C. Okasaki's type of binary random-access lists (Okasaki, 1998).

$$\begin{array}{lcl} \textbf{data}\ \mathit{Fork}\ a & = & \mathit{Fork}\ a\ a \\ \textbf{data}\ \mathit{Sequ}\ a & = & \mathit{Empty} \mid \mathit{Zero}\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \mid \mathit{One}\ a\ (\mathit{Sequ}\ (\mathit{Fork}\ a)) \end{array}$$

Since the type argument is changed in the recursive calls, *Sequ* is an example of a so-called nested or non-regular datatype (Bird & Meertens, 1998). Nested datatypes have recently received a great deal of attention since they can capture data-structural invariants in a way that regular datatypes cannot. For instance, *Sequ* captures the invariant that binary random-access lists are sequences of perfect binary leaf trees stored in increasing order of height. Though the type recursion is nested, the definition of the mapping function is entirely straightforward.

$$\begin{array}{lcl} \mathit{mapFork} & :: & \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{Fork}\ a_1 \to \mathit{Fork}\ a_2) \\ \mathit{mapFork}\ \mathit{mapa}\ (\mathit{Fork}\ v_1\ v_2) & = & \mathit{Fork}\ (\mathit{mapa}\ v_1)\ (\mathit{mapa}\ v_2) \\ \mathit{mapSequ} & :: & \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{Sequ}\ a_1 \to \mathit{Sequ}\ a_2) \\ \mathit{mapSequ}\ \mathit{mapa}\ \mathit{Empty} & = & \mathit{Empty} \\ \mathit{mapSequ}\ \mathit{mapa}\ (\mathit{Zero}\ \mathit{vs}) & = & \mathit{Zero}\ (\mathit{mapSequ}\ (\mathit{mapFork}\ \mathit{mapa})\ \mathit{vs}) \\ \mathit{mapSequ}\ \mathit{mapa}\ (\mathit{One}\ v\ \mathit{vs}) & = & \mathit{One}\ (\mathit{mapa}\ v)\ (\mathit{mapSequ}\ (\mathit{mapFork}\ \mathit{mapa})\ \mathit{vs}) \end{array}$$

Note that *mapSequ* requires polymorphic recursion (Mycroft, 1984): the recursive calls have type $\forall a_1\ a_2.(\mathit{Fork}\ a_1 \to \mathit{Fork}\ a_2) \to (\mathit{Sequ}\ (\mathit{Fork}\ a_1) \to \mathit{Sequ}\ (\mathit{Fork}\ a_2))$, which is a substitution instance of the declared type. Haskell 98 allows polymorphic recursion only if an explicit type signature is provided. The rationale behind this restriction is that type inference in the presence of polymorphic recursion is undecidable (Henglein, 1993).

Since *Sequ* is a nested type, it cannot be expressed as a fixpoint of a functor. However, it can be rephrased as a fixpoint of a higher-order functor (Bird & Meertens, 1998). Again, we will carry out the construction to generate examples of higher-order kinded datatypes. The higher-order base functor associated with *Sequ* is

$$\textbf{data}\ \mathit{SequF}\ s\ a \quad = \quad \mathit{Empty} \mid \mathit{Zero}\ (s\ (\mathit{Fork}\ a)) \mid \mathit{One}\ a\ (s\ (\mathit{Fork}\ a))\ .$$

Since *Sequ* has kind $\star \to \star$, its base functor has kind $(\star \to \star) \to (\star \to \star)$. The fixpoint operator for functors of this kind is given by

$$\textbf{newtype}\ \mathit{HFix}\ h\ a \quad = \quad \mathit{In}\ (h\ (\mathit{HFix}\ h)\ a)\ .$$

Since the fixpoint operator takes a second-order kinded type as argument, it has a third-order kind: $((\star \to \star) \to (\star \to \star)) \to (\star \to \star)$. Finally, we can define *Sequ* as the least fixpoint of *SequF*.

$$\textbf{type}\ \mathit{Sequ}' \quad = \quad \mathit{HFix}\ \mathit{SequF}$$

As a last stress test let us define a mapping function for Sequ'. As before we begin by defining mapping functions for the component types.

$$
\begin{aligned}
&\mathit{mapSequF} \quad :: \quad \forall s_1\ s_2.(\forall b_1\ b_2.(b_1 \to b_2) \to (s_1\ b_1 \to s_2\ b_2))\\
&\qquad\qquad\qquad\qquad \to \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{SequF}\ s_1\ a_1 \to \mathit{SequF}\ s_2\ a_2)\\
&\mathit{mapSequF}\ \mathit{maps}\ \mathit{mapa}\ \mathit{Empty}\\
&\qquad\qquad = \quad \mathit{Empty}\\
&\mathit{mapSequF}\ \mathit{maps}\ \mathit{mapa}\ (\mathit{Zero}\ \mathit{vs})\\
&\qquad\qquad = \quad \mathit{Zero}\ (\mathit{maps}\ (\mathit{mapFork}\ \mathit{mapa})\ \mathit{vs})\\
&\mathit{mapSequF}\ \mathit{maps}\ \mathit{mapa}\ (\mathit{One}\ v\ \mathit{vs})\\
&\qquad\qquad = \quad \mathit{One}\ (\mathit{mapa}\ v)\ (\mathit{maps}\ (\mathit{mapFork}\ \mathit{mapa})\ \mathit{vs})
\end{aligned}
$$

This example indicates why argument *maps* of kind $\star \to \star$ must be polymorphic: both calls of *maps* are instances of the declared type. In general, the argument mapping function may be applied to many different types. Admittedly, the type signature of *mapSequF* looks quite puzzling. However, we will see in a moment that it is fully determined by *SequF*'s kind. Even more daunting is the signature of *mapHFix*, which has rank 3. Unfortunately, neither GHC nor Hugs support rank-3 type signatures. Hence, the following code cannot be executed.

$$
\begin{aligned}
&\mathit{mapHFix} \quad :: \quad \forall h_1\ h_2.(\forall f_1\ f_2.(\forall c_1\ c_2.(c_1 \to c_2) \to (f_1\ c_1 \to f_2\ c_2))\\
&\qquad\qquad\qquad\qquad\qquad\qquad \to \forall b_1\ b_2.(b_1 \to b_2) \to (h_1\ f_1\ b_1 \to h_2\ f_2\ b_2))\\
&\qquad\qquad\qquad\qquad \to \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{HFix}\ h_1\ a_1 \to \mathit{HFix}\ h_2\ a_2)\\
&\mathit{mapHFix}\ \mathit{maph}\ \mathit{mapa}\ (\mathit{In}\ v)\\
&\qquad\qquad = \quad \mathit{In}\ (\mathit{maph}\ (\mathit{mapHFix}\ \mathit{maph})\ \mathit{mapa}\ v)
\end{aligned}
$$

Finally, applying *mapHFix* to *mapSequF* we obtain the desired function.

$$
\begin{aligned}
\mathit{mapSequ}' \quad &:: \quad \forall a_1\ a_2.(a_1 \to a_2) \to (\mathit{Sequ}'\ a_1 \to \mathit{Sequ}'\ a_2)\\
\mathit{mapSequ}' \quad &= \quad \mathit{mapHFix}\ \mathit{mapSequF}
\end{aligned}
$$

Now, let us define a polytypic version of *map*. The monotypic instances above already indicate that the type of the mapping function depends on the kind of the type index. In fact, the type of *map* can be defined by induction on the structure of kinds. A note on notation: we will write type and kind indices in angle brackets. Hence, $\mathit{map}\langle t :: \kappa\rangle$ denotes the application of the polytypic *map* to the type t of kind κ. We use essentially the same syntax both for polytypic values and for polykinded types. However, they are easily distinguished by their 'types', where the 'type' of kinds is given by the superkind '$\Box$'.

What is the type of *map* if the type-index has kind $\star$? For a manifest type, say, t the mapping function $\mathit{map}\langle t :: \star\rangle$ equals the identity function. Hence, its type is $t \to t$. In general, the mapping function $\mathit{map}\langle t :: \kappa\rangle$ has type $\mathit{Map}\langle\kappa\rangle\ t\ t$, where $\mathit{Map}\langle\kappa\rangle$ is defined as follows.

$$
\begin{aligned}
\mathit{Map}\langle\kappa :: \Box\rangle \quad &:: \quad \kappa \to \kappa \to \star\\
\mathit{Map}\langle\star\rangle\ t_1\ t_2 \quad &= \quad t_1 \to t_2\\
\mathit{Map}\langle\kappa_1 \to \kappa_2\rangle\ t_1\ t_2 \quad &= \quad \forall x_1\ x_2.\mathit{Map}\langle\kappa_1\rangle\ x_1\ x_2 \to \mathit{Map}\langle\kappa_2\rangle\ (t_1\ x_1)\ (t_2\ x_2)
\end{aligned}
$$

The first line of the definition is the so-called kind signature, which makes precise that $\mathit{Map}\langle\kappa :: \Box\rangle$ maps two types of kind κ to a manifest type. In the base case

$Map\langle\star\rangle\ t_1\ t_2$ equals the type of a conversion function. The inductive case has a very characteristic form, which we will encounter time and again. It specifies that a 'conversion function' between the type constructors t_1 and t_2 is a function that maps a conversion function between x_1 and x_2 to a conversion function between $t_1\ x_1$ and $t_2\ x_2$, for all possible instances of x_1 and x_2. Roughly speaking, $Map\langle\kappa_1 \to \kappa_2\rangle\ t_1\ t_2$ is the type of a 'conversion function'-transformer. It is not hard to see that the type signatures we have seen before are instances of this scheme. Furthermore, from the inductive definition above we can easily conclude that the rank of the type signature corresponds to the kind of the type index: the *map* for a third-order kinded type, for instance, has a rank-3 type signature.

How can we define the polytypic mapping function itself? It appears that the technique described in (Hinze, 2000) carries over to the polykinded case, that is, to define a polytypic value it suffices to give cases for primitive types, the unit type '1', sums '+', and products '×'. To be able to give polytypic definitions in a pointwise style, we treat 1, '+', and '×' as if they were given by the following datatype declarations.

$$\begin{array}{lcl}
\mathbf{data}\ 1 & = & () \\
\mathbf{data}\ a + b & = & \mathit{Inl}\ a \mid \mathit{Inr}\ b \\
\mathbf{data}\ a \times b & = & (a, b)
\end{array}$$

Assuming that we have only one primitive type, *Int*, the polytypic mapping function is given by

$$\begin{array}{lcl}
\mathit{map}\langle t :: \kappa\rangle & :: & \mathit{Map}\langle\kappa\rangle\ t\ t \\
\mathit{map}\langle 1\rangle\ () & = & () \\
\mathit{map}\langle \mathit{Int}\rangle\ i & = & i \\
\mathit{map}\langle +\rangle\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Inl}\ v) & = & \mathit{Inl}\ (\mathit{mapa}\ v) \\
\mathit{map}\langle +\rangle\ \mathit{mapa}\ \mathit{mapb}\ (\mathit{Inr}\ w) & = & \mathit{Inr}\ (\mathit{mapb}\ w) \\
\mathit{map}\langle \times\rangle\ \mathit{mapa}\ \mathit{mapb}\ (v, w) & = & (\mathit{mapa}\ v, \mathit{mapb}\ w)\ .
\end{array}$$

This straightforward definition contains all the ingredients needed to derive *map*s for arbitrary datatypes of arbitrary kinds. And, in fact, all the definitions we have seen before were automatically generated using a prototype implementation of the polytypic programming extension described in the subsequent sections. Finally, note that we can define *map* even more succinctly if we use a point-free style—as usual, the *map*s on sums and products are denoted $(+)$ and $(\times)$.

$$\begin{array}{lcl}
\mathit{map}\langle 1\rangle & = & \mathit{id} \\
\mathit{map}\langle \mathit{Int}\rangle & = & \mathit{id} \\
\mathit{map}\langle +\rangle\ \mathit{mapa}\ \mathit{mapb} & = & \mathit{mapa} + \mathit{mapb} \\
\mathit{map}\langle \times\rangle\ \mathit{mapa}\ \mathit{mapb} & = & \mathit{mapa} \times \mathit{mapb}
\end{array}$$

3 The simply typed lambda calculus as a type language

This section introduces kinds and types. The type system is essentially that of Haskell 98 (Peyton Jones & Hughes, 1999) smoothing away some of its irregularities.

Haskell offers one basic construct for defining new types: datatype declarations. In general, a **data** declaration has the following form.

$$\mathbf{data}\ T\ x_1\ \ldots\ x_m \quad = \quad C_1\ t_{11}\ \ldots\ t_{1m_1} \mid \cdots \mid C_n\ t_{n1}\ \ldots\ t_{nm_n}$$

This constructs combines no less than four different features: type abstraction, type recursion, n-ary sums, and n-ary products. The types on the right-hand side are built from type constants (that is, primitive type constructors), type variables, and type application. Thus, Haskell's type system essentially corresponds to the simply typed lambda calculus with kinds playing the rôle of types. This motivates the following definitions.

Definition 1
Kind terms are formed according to the following grammar.

$$K \quad ::= \quad \star \mid (K \to K)$$

As usual, we agree upon that '$\to$' associates to the right.

Type terms are built from type constants and type variables using type application and type abstraction. We annotate type constants and type variables with their kinds, that is, a type variable is a pair consisting of a name and a kind. If (s, κ) is a type constant or a type variable, we define $\mathit{kind}\ (s, \kappa) = \kappa$.

Definition 2
Given a set of kinded type constants $C \subseteq \Sigma^* \times K$ and a set of kinded type variables $X \subseteq \Sigma^* \times K$ type pseudo-terms are formed according to

$$T \quad ::= \quad C \mid X \mid (T\ T) \mid (\Lambda X.T) \mid \mu_K \mid \forall_K\ .$$

The choice of C is more or less arbitrary. For concreteness, we use

$$C \quad = \quad \{1 :: \star, \mathit{Int} :: \star, (+) :: \star \to \star \to \star, (\times) :: \star \to \star \to \star\}\ .$$

We write binary type constants infix and assume that they associate to the right. We agree upon that type application associates to the left and that type abstraction extends as far to the right as possible. Furthermore, we abbreviate $\Lambda x_1. \cdots \Lambda x_m.t$ by $\Lambda x_1\ \ldots\ x_m.t$. The set of pseudo-terms includes a family of fixpoint operators indexed by kind: $\mu_\star$ corresponds to *Fix* and $\mu_{\star\to\star}$ to *HFix*. Usually, we write $\mu x.t$ for $\mu_\kappa(\Lambda x.t)$ and similarly for the universal quantifier $\forall_\kappa$.

Definition 3
The set of legal type terms of kind κ, notation $T(\kappa)$, is defined by $T(\kappa) = \{t \in T \mid\ \vdash t :: \kappa\}$ where $\vdash t :: \kappa$ means that the statement $t :: \kappa$ is derivable using the rules depicted in Fig. 1. The set of monomorphic type terms of kind κ, which do not contain occurrences of $\forall$, is denoted $T^\circ(\kappa)$.

It is worth noting that since type constants and type variables are annotated with their kinds, we do not require a typing environment.

Given this type language we can easily translate datatype declarations into type terms. For instance, the type T defined by the schematic **data** declaration above

$$\frac{}{c :: kind\ c}\ (\text{T-CONST}) \qquad \frac{}{x :: kind\ x}\ (\text{T-VAR})$$

$$\frac{t_1 :: (\kappa_1 \to \kappa_2) \qquad t_2 :: \kappa_1}{(t_1\ t_2) :: \kappa_2}\ (\text{T-}\to\text{-ELIM}) \qquad \frac{t :: \kappa}{(\Lambda x.t) :: (kind\ x \to \kappa)}\ (\text{T-}\to\text{-INTRO})$$

$$\frac{}{\mu_\kappa :: (\kappa \to \kappa) \to \kappa}\ (\text{T-FIX}) \qquad \frac{}{\forall_\kappa :: (\kappa \to \star) \to \star}\ (\text{T-ALL})$$

Fig. 1. Kind rules.

is modelled by (we tacitly assume that the kinds of the type variables have been inferred)

$$\mu T.\Lambda x_1\ \ldots\ x_m.(t_{11} \times \cdots \times t_{1m_1}) + \cdots + (t_{n1} \times \cdots \times t_{nm_n})\ ,$$

where $t_1 \times \cdots \times t_0 = 1$. For simplicity, n-ary sums are reduced to binary sums and n-ary products to binary products.

Remark 2

The representation of regular types can be improved by applying lambda-dropping (Danvy, 1999) on the type level: if $\mu f.\Lambda a.t$ is regular, then it is equivalent to $\Lambda a.\mu b.t\ [f\ a := b]$ where $t\ [t_1 := t_2]$ denotes the type term, in which all occurrences of t_1 are replaced by t_2. For instance, the λ-dropped version of $\mu List.\Lambda a.1 + a \times List\ a$ is $\Lambda a.\mu b.\Lambda a.1 + a \times b$. Note that the λ-dropped and the λ-lifted version correspond to the two methods of modelling parameterized types given in Section 2: families of first-order fixpoints vs. higher-order fixpoints, see also (Bird & Paterson, 1999).

In Section 5 we require the following conversion rules for type terms.

Definition 4

The convertibility relation on type terms, denoted '$\leftrightarrow$', is given by the following axioms

$$(\Lambda x.t)\ u \quad \leftrightarrow \quad t\ [x := u] \qquad (\beta)$$

$$\mu t \quad \leftrightarrow \quad t\ (\mu t) \qquad (\mu)$$

plus rules for reflexivity, symmetry, transitivity, and congruence.

4 Defining polytypic values

The definition of a polytypic value consists of two parts: a type signature, which typically involves a polykinded type, and a set of equations, one for each type constant. Likewise, the definition of a polykinded type consists of two parts: a kind signature and one equation for kind $\star$. Interestingly, the equation for functional kinds need not be explicitly specified. It is inevitable because of the way type constructors of kind $\kappa_1 \to \kappa_2$ are specialized. We will return to this point in the following section.

In general, a polykinded type definition has the following schematic form.

$$
\begin{array}{lcl}
Poly\langle\kappa :: \Box\rangle & :: & \kappa \to \cdots \to \kappa \to \star \\
Poly\langle\star\rangle\ t_1\ \ldots\ t_n & = & \ldots \\
Poly\langle\kappa_1 \to \kappa_2\rangle\ t_1\ \ldots\ t_n & = & \forall x_1\ \ldots\ x_n . Poly\langle\kappa_1\rangle\ x_1\ \ldots\ x_n \\
 & & \qquad \to Poly\langle\kappa_2\rangle\ (t_1\ x_1)\ \ldots\ (t_n\ x_n)
\end{array}
$$

The kind signature makes precise that the kind-indexed type $Poly\langle\kappa :: \Box\rangle$ maps n types of kind κ to a manifest type (for $Map\langle\kappa :: \Box\rangle$ we had $n = 2$). The polytypic programmer merely has to fill out the right-hand side of the first equation.

Given the polykinded type a polytypic value definition takes on the following schematic form.

$$
\begin{array}{lcl}
poly\langle t :: \kappa\rangle & :: & Poly\langle\kappa\rangle\ t\ \ldots\ t \\
poly\langle 1\rangle & = & \ldots \\
poly\langle Int\rangle & = & \ldots \\
poly\langle +\rangle\ polya\ polyb & = & \ldots \\
poly\langle \times\rangle\ polya\ polyb & = & \ldots
\end{array}
$$

Again, the polytypic programmer has to fill out the right-hand sides. To be well-typed, the $poly\langle c :: \kappa\rangle$ instances must have type $Poly\langle\kappa\rangle\ c\ \ldots\ c$ as stated in the type signature.

5 Specializing polytypic values

This section is concerned with the specialization of polytypic values to concrete instances of datatypes. We have seen in Section 2 that the structure of each instance of $map\langle t\rangle$ rigidly follows the structure of t. Perhaps surprisingly, the intimate correspondence between the type and the value level holds not only for *map* but for all polytypic values. In fact, the process of specialization can be seen as an interpretation of the simply typed lambda calculus. The polytypic programmer specifies the interpretation of type constants. Given this information the meaning of a type term—that is, the specialization of a polytypic value—is fixed: type application is interpreted by value application, type abstraction by value abstraction, and type recursion by value recursion. Formally, the extension of *poly* to arbitrary type terms is given by

$$
\begin{array}{lcl}
poly\langle x\rangle & = & poly_x \\
poly\langle t_1\ t_2\rangle & = & (poly\langle t_1\rangle)\ (poly\langle t_2\rangle) \\
poly\langle \Lambda x . t\rangle & = & \lambda poly_x . poly\langle t\rangle \\
poly\langle \mu_\kappa\rangle & = & fix\ .
\end{array}
$$

Note that we consider only monomorphic types. The extension to polymorphic types is subject of future work. For the translation we use a simple variable naming convention, which obviates the need for an explicit environment. We agree upon that $poly\langle x\rangle$ is mapped to the variable $poly_x$. We often write $poly_x$ by concatenating the name of the polytypic value and the name of the type variable as in *mapa*. Of course, to avoid name capture we assume that $poly_x$ is distinct from variables introduced by the polytypic programmer.

$$\frac{}{p :: \mathit{type}\ p}\ (\textsc{const}) \qquad \frac{}{v :: \mathit{type}\ v}\ (\textsc{var})$$

$$\frac{e_1 :: (t_1 \to t_2) \qquad e_2 :: t_1}{(e_1\ e_2) :: t_2}\ (\to\text{-}\textsc{elim}) \qquad \frac{e :: t}{(\lambda v.e) :: (\mathit{type}\ v \to t)}\ (\to\text{-}\textsc{intro})$$

$$\frac{e :: (\forall x.t) \qquad u :: \mathit{kind}\ x}{(e\ u) :: t\,[x := u]}\ (\forall\text{-}\textsc{elim}) \qquad \frac{e :: t}{(\lambda x.e) :: (\forall x.t)}\ \begin{array}{l}x \text{ not free in the type}\\ \text{of a free variable of } e\end{array}\ (\forall\text{-}\textsc{intro})$$

$$\frac{}{\mathit{fix} :: \forall a.(a \to a) \to a}\ (\textsc{fix})$$

$$\frac{e :: t \qquad t \leftrightarrow u}{e :: u}\ (\textsc{conv})$$

Fig. 2. Typing rules.

The definition of *poly* makes precise that the specialization of a type constructor is a function. This explains why $\mathit{Poly}\langle\kappa_1 \to \kappa_2\rangle\ t_1\ \ldots\ t_n$ must necessarily be a function type. In the rest of this section we show that $\mathit{poly}\langle t\rangle$ is indeed well-typed. To this end we must first fix the target language we are compiling polytypic values to. Since we require first-class polymorphism, we will use a variant of the polymorphic lambda calculus (Girard, 1972), $F\omega$, augmented by a polymorphic fixpoint operator. Note that a similar language is also used as the internal language of the Glasgow Haskell Compiler (Peyton Jones, 1996).

Similar to the type language we annotate value constants and variables with their types. If (s, t) is a constant or a variable, we define $\mathit{type}\ (s, t) = t$.

Definition 5
Given a set of typed constants $P \subseteq \Sigma^* \times T(\star)$ and a set of typed variables $V \subseteq \Sigma^* \times T(\star)$ pseudo-terms are formed according to

$$E \quad ::= \quad P \mid V \mid (E\ E) \mid (\lambda V.E) \mid (E\ T) \mid (\lambda X.E) \mid \mathit{fix}\ .$$

Here, $e\ t$ denotes universal application and $\lambda x.e$ denotes universal abstraction. Note that we use the same syntax for value abstraction $\lambda v.e$ (here v is a value variable) and universal abstraction $\lambda x.e$ (here x is a type variable). To be able to define the $\mathit{poly}\langle c\rangle$ instances the set of value constants P should include suitable functions for each of the type constants in C.

Definition 6
The set of legal terms of type t, notation $E(t)$, is defined by $E(t) = \{\, e \in E \mid\ \vdash e :: t\,\}$ where $\vdash e :: t$ means that the statement $e :: t$ is derivable using the rules depicted in Fig. 2.

The type signature of a polytypic value determines the type for closed type indices. However, since the specialization is defined by induction on the structure of type terms, we must also explicate the type for type indices that contain free type variables. Take, for instance, the type term $\mathit{List}\ a$, in which a occurs free. The corresponding mapping function is $\mathit{mapList}\ \mathit{mapa}$, which has type $\mathit{List}\ a_1 \to \mathit{List}\ a_2$. In general, $\mathit{poly}\langle t :: \kappa\rangle$ has type $\mathit{Poly}\langle\kappa\rangle\ (t)_1\ \ldots\ (t)_n$ where $(t)_i$ denotes the type term t, in which every free type variable x has been replaced by x_i. To make this work we lay down that the value variable poly_x has type $\mathit{Poly}\langle \mathit{kind}\ x\rangle\ x_1\ \ldots\ x_n$ and that the x_i are fresh type variables associated with x. For instance, *mapa* has type $\mathit{Map}\langle\star\rangle\ a_1\ a_2 = a_1 \to a_2$. Given these prerequisites the fully typed extension of *poly* is defined by

$$
\begin{array}{lcl}
\mathit{poly}\langle t :: \kappa\rangle & :: & \mathit{Poly}\langle\kappa\rangle\ (t)_1\ \ldots\ (t)_n \\
\mathit{poly}\langle x\rangle & = & \mathit{poly}_x \\
\mathit{poly}\langle t_1\ t_2\rangle & = & (\mathit{poly}\langle t_1\rangle)\ (t_2)_1\ \ldots\ (t_2)_n\ (\mathit{poly}\langle t_2\rangle) \\
\mathit{poly}\langle \Lambda x.t\rangle & = & \lambda x_1\ \ldots\ x_n.\lambda \mathit{poly}_x.\mathit{poly}\langle t\rangle \\
\mathit{poly}\langle \mu_\kappa\rangle & = & \lambda x_1\ \ldots\ x_n.\lambda \mathit{poly}_x.\mathit{fix}\ (\mathit{Poly}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n)) \\
 & & \qquad\qquad\qquad\qquad (\mathit{poly}_x\ (\mu x_1)\ \ldots\ (\mu x_n))\ .
\end{array}
$$

Theorem 1

If $\mathit{poly}\langle c\rangle \in E(\mathit{Poly}\langle \mathit{kind}\ c\rangle\ c\ \ldots\ c)$ for all type constants $c \in C$, then $\mathit{poly}\langle u\rangle \in E(\mathit{Poly}\langle\kappa\rangle\ (u)_1\ \ldots\ (u)_n)$ for all type terms $u \in T^\circ(\kappa)$.

Proof

We use induction on the derivation of $u :: \kappa$. We content ourselves with one illustrative case. **Case** $u = \mu_\kappa$: Note that $\mathit{kind}\ x_i = \mathit{kind}\ x = \kappa \to \kappa$ (the kinds of the type variables do not appear explicitly in the definition of *poly*). By convention, we have $\mathit{poly}_x :: \mathit{Poly}\langle\kappa\to\kappa\rangle\ x_1\ \ldots\ x_n$. Consequently,

$$
\begin{array}{l}
\mathit{poly}_x\ (\mu x_1)\ \ldots\ (\mu x_n) \\
\quad ::\ \ \mathit{Poly}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n) \to \mathit{Poly}\langle\kappa\rangle\ (x_1\ (\mu x_1))\ \ldots\ (x_n\ (\mu x_n))
\end{array}
$$

and since $\mu t \leftrightarrow t\ (\mu t)$ and $\mathit{fix} :: \forall a.(a \to a) \to a$ we obtain

$$
\begin{array}{l}
\lambda x_1\ \ldots\ x_n.\lambda \mathit{poly}_x.\mathit{fix}\ (\mathit{Poly}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n))\ (\mathit{poly}_x\ (\mu x_1)\ \ldots\ (\mu x_n)) \\
\quad ::\ \ \forall x_1\ \ldots\ x_n.\mathit{Poly}\langle\kappa\to\kappa\rangle\ x_1\ \ldots\ x_n \to \mathit{Poly}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n)
\end{array}
$$

as desired. $\Box$

Let us conclude the section by noting a trivial consequence of the specialization. Since the structure of types is reflected on the value level, we have $\mathit{poly}\langle\Lambda a.f\ (g\ a)\rangle = \lambda \mathit{poly}_a.\mathit{poly}\langle f\rangle\ ((\mathit{poly}\langle g\rangle)\ \mathit{poly}_a)$. Writing composition as usual this implies, in particular, that $\mathit{map}\langle f \cdot g\rangle = \mathit{map}\langle f\rangle \cdot \mathit{map}\langle g\rangle$. Perhaps surprisingly, this relationship holds for all polytypic values not only for mapping functions.

6 Examples

This section presents further examples of polytypic values with polykinded types.

6.1 Polytypic equality

The equality function *equal* serves as a typical example of a polytypic value. The polykinded equality type is fairly straightforward: for a manifest type $\mathit{equal}\langle t\rangle$ has type $t \to t \to Bool$, which determines the following definition.

$$
\begin{array}{lcl}
Equal\langle\kappa :: \Box\rangle & :: & \kappa \to \star \\
Equal\langle\star\rangle\ t & = & t \to t \to Bool \\
Equal\langle\kappa_1 \to \kappa_2\rangle\ t & = & \forall x. Equal\langle\kappa_1\rangle\ x \to Equal\langle\kappa_2\rangle\ (t\ x)
\end{array}
$$

For ease of reference we will always list the equation for functional kinds even though it is fully determined by the theory. Assuming that a suitable equality function for *Int* is available, the polytypic equality function can be defined as follows (for reasons of readability we use a Haskell-like notation rather than $F\omega$).

$$
\begin{array}{lcl}
equal\langle t :: \kappa\rangle & :: & Equal\langle\kappa\rangle\ t \\
equal\langle 1\rangle\ ()\ () & = & True \\
equal\langle Int\rangle\ i\ j & = & equalInt\ i\ j \\
equal\langle +\rangle\ equala\ equalb\ (Inl\ v_1)\ (Inl\ v_2) & = & equala\ v_1\ v_2 \\
equal\langle +\rangle\ equala\ equalb\ (Inl\ v_1)\ (Inr\ w_2) & = & False \\
equal\langle +\rangle\ equala\ equalb\ (Inr\ w_1)\ (Inl\ v_2) & = & False \\
equal\langle +\rangle\ equala\ equalb\ (Inr\ w_1)\ (Inr\ w_2) & = & equalb\ w_1\ w_2 \\
equal\langle \times\rangle\ equala\ equalb\ (v_1, w_1)\ (v_2, w_2) & = & equala\ v_1\ v_2 \wedge equalb\ w_1\ w_2
\end{array}
$$

Now, since *equal* has a kind-indexed type we can also specialize it for, say, unary type constructors.

$$
equal\langle f :: \star \to \star\rangle \quad :: \quad \forall x.(x \to x \to Bool) \to (f\ x \to f\ x \to Bool)
$$

This gives us an extra degree of flexibility: $equal\langle f\rangle\ op\ v\ w$ checks whether corresponding elements in v and w are related by op. Of course, op need not be an equality operator. PolyLib (Jansson & Jeuring, 1998) defines an analogous function but with a more general type:

$$
pequal\langle f :: \star \to \star\rangle \quad :: \quad \forall x_1\ x_2.(x_1 \to x_2 \to Bool) \to (f\ x_1 \to f\ x_2 \to Bool)\ .
$$

Here, the element types need not be identical. And, in fact, $equal\langle t :: \kappa\rangle$ can be assigned the more general type $PEqual\langle\kappa\rangle\ t\ t$ given by

$$
\begin{array}{lcl}
PEqual\langle\kappa :: \Box\rangle & :: & \kappa \to \kappa \to \star \\
PEqual\langle\star\rangle\ t_1\ t_2 & = & t_1 \to t_2 \to Bool \\
PEqual\langle\kappa_1 \to \kappa_2\rangle\ t_1\ t_2 & = & \forall x_1\ x_2. PEqual\langle\kappa_1\rangle\ x_1\ x_2 \to PEqual\langle\kappa_2\rangle\ (t_1\ x_1)\ (t_2\ x_2)\ ,
\end{array}
$$

which gives us an even greater degree of flexibility.

6.2 Mapping and zipping functions

In Section 2 we have seen how to define mapping functions for types of arbitrary kinds. Interestingly, the polytypic map subsumes so-called higher-order maps. A higher-order functor operates on a functor category, which has as objects functors and as arrows natural transformations. In Haskell we can model natural transformations by polymorphic functions.

$$
\mathbf{type}\ f_1 \mathbin{\dot{\to}} f_2 \quad = \quad \forall a. f_1\ a \to f_2\ a
$$

A natural transformation between functors F_1 and F_2 is a polymorphic function of type $F_1 \dot{\to} F_2$. A higher-order functor H then consists of a type constructor of kind $(\star \to \star) \to (\star \to \star)$, such as $SequF$, and an associated mapping function of type $(f_1 \dot{\to} f_2) \to (H\ f_1 \dot{\to} H\ f_2)$. Now, the polytypic map gives us a function of type

$$\begin{aligned} map\langle H\rangle \ &:: \ \forall f_1\ f_2.(\forall b_1\ b_2.(b_1 \to b_2) \to (f_1\ b_1 \to f_2\ b_2)) \\ &\qquad \to (\forall a_1\ a_2.(a_1 \to a_2) \to (H\ f_1\ a_1 \to H\ f_2\ a_2)) \ . \end{aligned}$$

Given a natural transformation α of type $f_1 \dot{\to} f_2$ there are basically two alternatives for constructing a function of type $\forall b_1\ b_2.(b_1 \to b_2) \to (f_1\ b_1 \to f_2\ b_2)$: $\lambda m.\alpha \cdot map\langle f_1\rangle\ m$ or $\lambda m.map\langle f_2\rangle\ m \cdot \alpha$. The naturality of α, however, implies that both alternatives are equal. Consequently, the higher-order map is given by

$$\begin{aligned} &hmap\langle h :: (\star \to \star) \to (\star \to \star)\rangle &&:: \ \forall f_1\ f_2.(f_1 \dot{\to} f_2) \to (h\ f_1 \dot{\to} h\ f_2) \\ &hmap\langle h\rangle\ \alpha &&= \ map\langle h\rangle\ (\lambda m.\alpha \cdot map\langle f_1\rangle\ m)\ id \ . \end{aligned}$$

Using definitions similar to the one in Section 2 we can also implement embedding-projection maps of type $MapE\langle\star\rangle\ t_1\ t_2 = (t_1 \to t_2, t_2 \to t_1)$, monadic maps (Fokkinga, 1994; Meijer & Jeuring, 1995) of type $MapM\langle\star\rangle\ t_1\ t_2 = t_1 \to M\ t_2$ for some monad M, and arrow maps (Jansson & Jeuring, 1999) of type $MapA\langle\star\rangle\ t_1\ t_2 = t_1 \rightsquigarrow t_2$ for some arrow type $(\rightsquigarrow)$.

Closely related to mapping functions are zipping functions. A binary zipping function takes two structures of the same shape and combines them into a single structure. For instance, the list *zip* takes a function of type $a_1 \to a_2 \to a_3$, two lists of type $List\ a_1$ and $List\ a_2$ and applies the function to corresponding elements producing a list of type $List\ a_3$. The type of the polytypic *zip* is essentially a three parameter variant of *Map*.

$$\begin{aligned} &Zip\langle\kappa :: \Box\rangle &&:: \ \kappa \to \kappa \to \kappa \to \star \\ &Zip\langle\star\rangle\ t_1\ t_2\ t_3 &&= \ t_1 \to t_2 \to t_3 \\ &Zip\langle\kappa_1 \to \kappa_2\rangle\ t_1\ t_2\ t_3 &&= \ \forall x_1\ x_2\ x_3.Zip\langle\kappa_1\rangle\ x_1\ x_2\ x_3 \to Zip\langle\kappa_2\rangle\ (t_1\ x_1)\ (t_2\ x_2)\ (t_3\ x_3) \end{aligned}$$

The definition of *zip* is similar to that of *equal*.

$$\begin{aligned} &zip\langle t :: \kappa\rangle &&:: \ Zip\langle\kappa\rangle\ t\ t\ t \\ &zip\langle 1\rangle\ ()\ () &&= \ () \\ &zip\langle Int\rangle\ i\ j &&= \ \mathbf{if}\ equalInt\ i\ j\ \mathbf{then}\ i\ \mathbf{else}\ undefined \\ &zip\langle +\rangle\ zipa\ zipb\ (Inl\ v_1)\ (Inl\ v_2) &&= \ Inl\ (zipa\ v_1\ v_2) \\ &zip\langle +\rangle\ zipa\ zipb\ (Inl\ v_1)\ (Inr\ w_2) &&= \ undefined \\ &zip\langle +\rangle\ zipa\ zipb\ (Inr\ w_1)\ (Inl\ v_2) &&= \ undefined \\ &zip\langle +\rangle\ zipa\ zipb\ (Inr\ w_1)\ (Inr\ w_2) &&= \ Inr\ (zipb\ w_1\ w_2) \\ &zip\langle \times\rangle\ zipa\ zipb\ (v_1, w_1)\ (v_2, w_2) &&= \ (zipa\ v_1\ v_2, zipb\ w_1\ w_2) \end{aligned}$$

Note that the result of *zip* is a partial structure if the two arguments have not the same shape. Alternatively, one can define a zipping function of type $Zip\langle\star\rangle\ t_1\ t_2\ t_3 = t_1 \to t_2 \to Maybe\ t_3$, which uses the exception monad *Maybe* to signal incompatibility of the argument structures, see (Hinze, 1999).

6.3 Reductions

A reduction or a crush (Meertens, 1996) is a polytypic function that collapses a structure of values of type x into a single value of type x. This section explains how to define reductions that work for all types of all kinds. To illustrate the main idea let us start with three motivating examples. The first one is a function that counts the number of values of type *Int* within a given structure of some type. Here is the type of the polytypic counter

$$\begin{array}{lcl}
\mathit{Count}\langle \kappa :: \Box \rangle & :: & \kappa \to \star \\
\mathit{Count}\langle \star \rangle\ t & = & t \to \mathit{Int} \\
\mathit{Count}\langle \kappa_1 \to \kappa_2 \rangle\ t & = & \forall x. \mathit{Count}\langle \kappa_1 \rangle\ x \to \mathit{Count}\langle \kappa_2 \rangle\ (t\ x)
\end{array}$$

and here is its definition.

$$\begin{array}{lcl}
\mathit{count}\langle t :: \kappa \rangle & :: & \mathit{Count}\langle \kappa \rangle\ t \\
\mathit{count}\langle 1 \rangle\ () & = & 0 \\
\mathit{count}\langle \mathit{Int} \rangle\ i & = & 1 \\
\mathit{count}\langle + \rangle\ \mathit{counta}\ \mathit{countb}\ (\mathit{Inl}\ v) & = & \mathit{counta}\ v \\
\mathit{count}\langle + \rangle\ \mathit{counta}\ \mathit{countb}\ (\mathit{Inr}\ w) & = & \mathit{countb}\ w \\
\mathit{count}\langle \times \rangle\ \mathit{counta}\ \mathit{countb}\ (v, w) & = & \mathit{counta}\ v + \mathit{countb}\ w
\end{array}$$

Next, let us consider a slight variation: the function $\mathit{size}\langle t \rangle$ defined below is identical to $\mathit{count}\langle t \rangle$ except for $t = \mathit{Int}$, in which case *size* also returns 0.

$$\begin{array}{lcl}
\mathit{size}\langle t :: \kappa \rangle & :: & \mathit{Count}\langle \kappa \rangle\ t \\
\mathit{size}\langle 1 \rangle\ () & = & 0 \\
\mathit{size}\langle \mathit{Int} \rangle\ i & = & 0 \\
\mathit{size}\langle + \rangle\ \mathit{sizea}\ \mathit{sizeb}\ (\mathit{Inl}\ v) & = & \mathit{sizea}\ v \\
\mathit{size}\langle + \rangle\ \mathit{sizea}\ \mathit{sizeb}\ (\mathit{Inr}\ w) & = & \mathit{sizeb}\ w \\
\mathit{size}\langle \times \rangle\ \mathit{sizea}\ \mathit{sizeb}\ (v, w) & = & \mathit{sizea}\ v + \mathit{sizeb}\ w
\end{array}$$

It is not hard to see that $\mathit{size}\langle t \rangle\ v$ returns 0 for all types t of kind $\star$ (provided v is fully defined). So one might be led to conclude that *size* is not a very useful function. This conclusion is, however, too rash since *size* can also be parameterized by type constructors. For instance, for unary type constructors *size* has type

$$\mathit{size}\langle f :: \star \to \star \rangle \quad :: \quad \forall x. (x \to \mathit{Int}) \to (f\ x \to \mathit{Int})$$

Now, if we pass the identity function to *size*, we obtain a function that sums up a structure of integers. Another viable choice is *const* 1; this yields a function of type $\forall x. f\ x \to \mathit{Int}$ that counts the number of values of type x in a given structure of type $f\ x$.

$$\begin{array}{lcl}
\mathit{fsum}\langle f :: \star \to \star \rangle & :: & f\ \mathit{Int} \to \mathit{Int} \\
\mathit{fsum}\langle f \rangle & = & \mathit{size}\langle f \rangle\ \mathit{id} \\
\mathit{fsize}\langle f :: \star \to \star \rangle & :: & \forall x. f\ x \to \mathit{Int} \\
\mathit{fsize}\langle f \rangle & = & \mathit{size}\langle f \rangle\ (\mathit{const}\ 1)
\end{array}$$

Using a similar approach we can also flatten a structure into a list of elements.

$$
\begin{array}{lcl}
\mathit{fsum}\langle f :: \star \to \star \rangle & :: & f\ \mathit{Int} \to \mathit{Int} \\
\mathit{fsum}\langle f \rangle & = & \mathit{reduce}\langle f \rangle\ 0\ (+)\ \mathit{id} \\
\mathit{fsize}\langle f :: \star \to \star \rangle & :: & \forall x. f\ x \to \mathit{Int} \\
\mathit{fsize}\langle f \rangle & = & \mathit{reduce}\langle f \rangle\ 0\ (+)\ (\mathit{const}\ 1) \\
\mathit{fand}\langle f :: \star \to \star \rangle & :: & f\ \mathit{Bool} \to \mathit{Bool} \\
\mathit{fand}\langle f \rangle & = & \mathit{reduce}\langle f \rangle\ \mathit{True}\ (\wedge)\ \mathit{id} \\
\mathit{fall}\langle f :: \star \to \star \rangle & :: & \forall x. (x \to \mathit{Bool}) \to (f\ x \to \mathit{Bool}) \\
\mathit{fall}\langle f \rangle\ p & = & \mathit{reduce}\langle f \rangle\ \mathit{True}\ (\wedge)\ p \\
\mathit{fflatten}\langle f :: \star \to \star \rangle & :: & \forall x. f\ x \to \mathit{List}\ x \\
\mathit{fflatten}\langle f \rangle & = & \mathit{reduce}\langle f \rangle\ \mathit{Nil}\ (+\!\!+)\ \mathit{wrap} \\
\mathit{gflatten}\langle g :: \star \to \star \to \star \rangle & :: & \forall x\ y. g\ x\ y \to \mathit{List}\ (x + y) \\
\mathit{gflatten}\langle g \rangle & = & \mathit{reduce}\langle g \rangle\ \mathit{Nil}\ (+\!\!+)\ (\mathit{wrap} \cdot \mathit{Inl})\ (\mathit{wrap} \cdot \mathit{Inr})
\end{array}
$$

Fig. 3. Examples of reductions.

The type of the polytypic flattening function

$$
\begin{array}{lcl}
\mathit{Flatten}\langle \kappa :: \Box \rangle & :: & \kappa \to \star \to \star \\
\mathit{Flatten}\langle \star \rangle\ t\ z & = & t \to \mathit{List}\ z \\
\mathit{Flatten}\langle \kappa_1 \to \kappa_2 \rangle\ t\ z & = & \forall x. \mathit{Flatten}\langle \kappa_1 \rangle\ x\ z \to \mathit{Flatten}\langle \kappa_2 \rangle\ (t\ x)\ z
\end{array}
$$

makes use of a simple extension: $\mathit{Flatten}\langle \kappa \rangle$ takes an additional type parameter, z, that is passed unchanged to the base case. One can safely think of z as a global type parameter. The code for *flatten* is similar to the code for *size*.

$$
\begin{array}{lcl}
\mathit{flatten}\langle t :: \kappa \rangle & :: & \forall z. \mathit{Flatten}\langle \kappa \rangle\ t\ z \\
\mathit{flatten}\langle 1 \rangle\ () & = & \mathit{Nil} \\
\mathit{flatten}\langle \mathit{Int} \rangle\ i & = & \mathit{Nil} \\
\mathit{flatten}\langle + \rangle\ \mathit{flattena}\ \mathit{flattenb}\ (\mathit{Inl}\ v) & = & \mathit{flattena}\ v \\
\mathit{flatten}\langle + \rangle\ \mathit{flattena}\ \mathit{flattenb}\ (\mathit{Inr}\ w) & = & \mathit{flattenb}\ w \\
\mathit{flatten}\langle \times \rangle\ \mathit{flattena}\ \mathit{flattenb}\ (v, w) & = & \mathit{flattena}\ v +\!\!+ \mathit{flattenb}\ w
\end{array}
$$

The function $(+\!\!+) :: \forall a. \mathit{List}\ a \to \mathit{List}\ a \to \mathit{List}\ a$ used in the last equation concatenates two lists. As before, *flatten* is pointless for types but useful for type constructors.

$$
\begin{array}{lcl}
\mathit{fflatten}\langle f :: \star \to \star \rangle & :: & \forall a. f\ a \to \mathit{List}\ a \\
\mathit{fflatten}\langle f \rangle & = & \mathit{flatten}\langle f \rangle\ \mathit{wrap}\ \textbf{where}\ \mathit{wrap}\ v = \mathit{Cons}\ v\ \mathit{Nil}
\end{array}
$$

The definitions of *size* and *flatten* exhibit a common pattern: the elements of a base type are replaced by a constant (0 and *Nil*, respectively) and the pair constructor is replaced by a binary operator ($(+)$ and $(+\!\!+)$, respectively). The polytypic function *reduce* abstracts away from these particularities.

$$
\begin{array}{lcl}
\mathit{Red}\langle\kappa :: \Box\rangle & :: & \kappa \to \star \to \star \\
\mathit{Red}\langle\star\rangle\ t\ z & = & t \to z \\
\mathit{Red}\langle\kappa_1 \to \kappa_2\rangle\ t\ z & = & \forall x.\mathit{Red}\langle\kappa_1\rangle\ x\ z \to \mathit{Red}\langle\kappa_2\rangle\ (t\ x)\ z \\
\mathit{reduce}\langle t :: \kappa\rangle & :: & \forall z.z \to (z \to z \to z) \to \mathit{Red}\langle\kappa\rangle\ t\ z \\
\mathit{reduce}\langle t\rangle\ e\ (\oplus) & = & \mathit{red}\langle t\rangle \\
\quad \mathbf{where} & & \\
\quad \mathit{red}\langle 1\rangle\ () & = & e \\
\quad \mathit{red}\langle \mathit{Int}\rangle\ i & = & e \\
\quad \mathit{red}\langle +\rangle\ \mathit{reda}\ \mathit{redb}\ (\mathit{Inl}\ v) & = & \mathit{reda}\ v \\
\quad \mathit{red}\langle +\rangle\ \mathit{reda}\ \mathit{redb}\ (\mathit{Inr}\ w) & = & \mathit{redb}\ w \\
\quad \mathit{red}\langle \times\rangle\ \mathit{reda}\ \mathit{redb}\ (v, w) & = & \mathit{reda}\ v \oplus \mathit{redb}\ w
\end{array}
$$

Note that we can define *red* even more succinctly using a point-free style.

$$
\begin{array}{lcl}
\mathit{red}\langle 1\rangle & = & \mathit{const}\ e \\
\mathit{red}\langle \mathit{Int}\rangle & = & \mathit{const}\ e \\
\mathit{red}\langle +\rangle\ \mathit{reda}\ \mathit{redb} & = & \mathit{reda}\ \triangledown\ \mathit{redb} \\
\mathit{red}\langle \times\rangle\ \mathit{reda}\ \mathit{redb} & = & \mathit{uncurry}\ (\oplus) \cdot (\mathit{reda} \times \mathit{redb})
\end{array}
$$

Here, $(\triangledown)$ is the so-called junction operator (Backhouse *et al.*, 1999). The type of $\mathit{reduce}\langle f\rangle$ where f is a unary type constructor is quite general.

$$
\mathit{reduce}\langle f :: \star \to \star\rangle \quad :: \quad \forall z.z \to (z \to z \to z) \to (\forall x.(x \to z) \to (f\ x \to z))
$$

Fig. 3 lists some typical applications of $\mathit{reduce}\langle f\rangle$ and $\mathit{reduce}\langle g\rangle$ where g is a binary type constructor. Further examples can be found, for instance, in (Meertens, 1996) and (Jansson & Jeuring, 1998).

7 Properties of polytypic values

This section investigates another important aspect of polytypism: polytypic reasoning. The section is structured as follows. Section 7.1 shows how to generalize the functorial laws to datatypes of arbitrary kinds and explains how to prove these generalizations correct. In Section 7.2 we develop a general framework for conducting polytypic proofs. This framework is then illustrated in Section 7.3 by several examples.

7.1 Functorial laws

To classify as a functor the mapping function of a unary type constructor must satisfy the so-called functorial laws:

$$
\begin{array}{lcl}
\mathit{map}\langle f\rangle\ \mathit{id} & = & \mathit{id} \\
\mathit{map}\langle f\rangle\ (\varphi \cdot \psi) & = & \mathit{map}\langle f\rangle\ \varphi \cdot \mathit{map}\langle f\rangle\ \psi\ ,
\end{array}
$$

that is, $map\langle f\rangle$ preserves identity and composition. If the type constructor is binary, the functor laws take the form

$$\begin{aligned} map\langle g\rangle\ id\ id &= id \\ map\langle g\rangle\ (\varphi_1 \cdot \psi_1)\ (\varphi_2 \cdot \psi_2) &= map\langle g\rangle\ \varphi_1\ \varphi_2 \cdot map\langle g\rangle\ \psi_1\ \psi_2\ . \end{aligned}$$

How can we generalize these laws to datatypes of arbitrary kinds? Since $map\langle t\rangle$ has a kind-indexed type, it is reasonable to expect that the functorial properties are indexed by kinds, as well. So, what form do the laws take if the type index is a manifest type, say, t? In this case $map\langle t\rangle$ does not preserve identity; it *is* the identity.

$$\begin{aligned} map\langle t\rangle &= id \\ map\langle t\rangle &= map\langle t\rangle \cdot map\langle t\rangle \end{aligned}$$

The pendant of the second law states that $map\langle t\rangle$ is idempotent (which is a simple consequence of the first law). Given this base case the generalization to arbitrary kinds is within reach. The polytypic version of the first functorial law states that $map\langle t\rangle \in \mathcal{I}\langle\kappa\rangle$ for all types $t \in T^{\circ}(\kappa)$, where $\mathcal{I}\langle\kappa\rangle$ is given by

$$\begin{aligned} \varphi \in \mathcal{I}\langle\star\rangle &\equiv \varphi = id \\ \varphi \in \mathcal{I}\langle\kappa_1 \to \kappa_2\rangle &\equiv \forall v\,\textbf{.}\, v \in \mathcal{I}\langle\kappa_1\rangle \supset \varphi\ v \in \mathcal{I}\langle\kappa_2\rangle\ . \end{aligned}$$

The relation $\mathcal{I}$ is a so-called unary logical relation (Mitchell, 1996). The second clause of the definition is characteristic for logical relations; it guarantees that the relation is closed under type application and type abstraction. Similarly, the polytypic version of the second functorial law states that $(map\langle t\rangle, map\langle t\rangle, map\langle t\rangle) \in \mathcal{C}\langle\kappa\rangle$ for all types $t \in T^{\circ}(\kappa)$, where $\mathcal{C}\langle\kappa\rangle$ is given by

$$\begin{aligned} (\varphi_1, \varphi_2, \varphi_3) \in \mathcal{C}\langle\star\rangle &\equiv \varphi_1 \cdot \varphi_2 = \varphi_3 \\ (\varphi_1, \varphi_2, \varphi_3) \in \mathcal{C}\langle\kappa_1 \to \kappa_2\rangle &\equiv \forall v_1\ v_2\ v_3\,\textbf{.}\,(v_1, v_2, v_3) \in \mathcal{C}\langle\kappa_1\rangle \\ &\qquad \supset (\varphi_1\ v_1, \varphi_2\ v_2, \varphi_3\ v_3) \in \mathcal{C}\langle\kappa_2\rangle\ . \end{aligned}$$

Note again the characteristic form of the second clause. It expresses that functions are related if related arguments are taken to related results. It is not hard to see that the monotypic functorial laws are instances of these polytypic laws.

Turning to the proof of the polytypic laws we must show (1) that $map\langle c\rangle \in \mathcal{I}\langle kind\ c\rangle$ and $(map\langle c\rangle, map\langle c\rangle, map\langle c\rangle) \in \mathcal{C}\langle kind\ c\rangle$ for all type constants $c \in C$ and (2) that $fix \in \mathcal{I}\langle\kappa\rangle$ and $(fix, fix, fix) \in \mathcal{C}\langle\kappa\rangle$ for all $\kappa \in K$. The first part follows directly from the functorial laws of $(+)$ and $(\times)$. The second part is more delicate. The usual approach is to impose two further conditions on logical relations: they must be pointed and chain complete[1]—we tacitly assume that we are working in a CPO setting. Then one can invoke fixpoint induction to show that fixpoint operators are logically related. Now, $\mathcal{C}$ satisfies these requirements; $\mathcal{I}$, however, does not since it fails to be pointed (clearly, $\bot = id$ does not hold in general). We can only prove the weaker statement $\varphi \in \mathcal{I}\langle\star\rangle \equiv \varphi \leqslant id$.

[1] A relation $\mathcal{R}$ is pointed if $\bot \in \mathcal{R}$; it is chain complete if $S \subseteq \mathcal{R} \supset \bigsqcup S \in \mathcal{R}$ for every chain S.

Let us take a closer look at the reason for this failure. To this end consider the type $Tree = \mu\, TreeF$ with $TreeF = \Lambda a.1 + a \times a$. The associated mapping function is $mapTree = fix\ mapTreeF$ with $mapTreeF = \lambda mapa.id + mapa \times mapa$. Now, it is fairly obvious that we cannot prove $mapTree = id$ by fixpoint induction. In a sense, the problem is that equality is only 'weakly typed': we must prove $\varphi\ v = v$ for all $v :: Tree$. Clearly, the approximations of $mapTree$, $mapTreeF^i\ \bot$, do not satisfy this strong condition. However, $mapTreeF^i\ \bot\ v = v$ holds for all elements $v :: TreeF^i\ 0$. Here, 0 is the 'empty' type, which contains $\bot$ as the single element. In other words, approximations on the value level and approximations on the type level should go hand in hand. This can be accomplished by parameterizing logical relations with the types involved: $\varphi \in \mathcal{I}\langle\star\rangle\ t \equiv \varphi = id :: t \to t$. Given this modification it is not hard to show that $\mathcal{I}\langle\star\rangle$ is pointed, that is, $\bot \in \mathcal{I}\langle\star\rangle\ 0$. Now we can invoke a polytypic version of Scott and de Bakker's fixpoint induction to conclude that $fix \in \mathcal{I}\langle\kappa\rangle\ \mu$ for all $\kappa \in K$. Polytypic fixpoint induction can be stated as the following inference rule:

$$\frac{\bot \in P\langle 0\rangle \qquad \forall x\,.\,\forall v\,.\,v \in P\langle x\rangle \supset \varphi\ v \in P\langle t\ x\rangle}{fix\ \varphi \in P\langle \mu t\rangle}\ (\textsc{fp-induction})\ ,$$

where P is a type-indexed chain-complete predicate. Here, chain completeness means that whenever $\varphi^i\ \bot \in P\langle t^i\ 0\rangle$ for all $i \in \mathbb{N}$ then $fix\ \varphi \in P\langle \mu t\rangle$.

7.2 Logical relations

Logical relations were originally developed for the study of typed lambda calculi. The interested reader is referred to J. Mitchell's textbook (Mitchell, 1996) for a comprehensive treatment of logical relations. The basic idea is as follows. Say, we are given two models of the simply typed lambda calculus. Then the Basic Lemma of logical relations establishes that the meaning of a term in one model is logically related to its meaning in the other model (provided the meanings of the constants are logically related).

We have discussed in Section 5 that the specialization of a polytypic value can be seen as an interpretation of the simply typed lambda calculus. Now, assume that we have two polytypic values $poly_1$ and $poly_2$. A suitable adaptation of the Basic Lemma shows that the meaning of $poly_1\langle t\rangle$ is logically related to the meaning of $poly_2\langle t\rangle$. For presenting the Basic Lemma we will use a 'semantic version' of $poly$, where the meaning of $poly\langle c\rangle$ is an element in some cpo.

$$\begin{array}{lcl} poly\langle x\rangle\ \eta & = & \eta\ x \\ poly\langle t_1\ t_2\rangle\ \eta & = & (poly\langle t_1\rangle\ \eta)\ (poly\langle t_2\rangle\ \eta) \\ poly\langle \Lambda x.t\rangle\ \eta & = & \lambda v.poly\langle t\rangle\ (\eta(x := v)) \\ poly\langle \mu_\kappa\rangle\ \eta & = & fix \end{array}$$

Here, η is an environment that maps type variables to values, and $\eta(x := v)$ is syntax for extending the environment η by the binding $x := v$. It is understood that the constructions on the right-hand side denote semantic values, for instance,

$\lambda v.poly\langle t\rangle\ (\eta(x := v))$ denotes a function mapping v to $poly\langle t\rangle\ (\eta(x := v))$. For closed type terms we write $poly\langle t\rangle$ instead of $poly\langle t\rangle\ \eta$.

In presenting logical relations we will restrict ourselves to the binary case. The extension to the n-ary case is, however, entirely straightforward.

Definition 7

A binary logical relation $\mathcal{R}$ over $\mathcal{D}_1$ and $\mathcal{D}_2$ is a kind-indexed relation such that

1. $\mathcal{R}\langle\kappa\rangle\ t_1\ \ldots\ t_n \subseteq \mathcal{D}_1\langle\kappa\rangle\ t_1\ \ldots\ t_n \times \mathcal{D}_2\langle\kappa\rangle\ t_1\ \ldots\ t_n$, where $\mathcal{D}_1$ and $\mathcal{D}_2$ are kind-indexed types,
2. $$\begin{aligned} &(\varphi_1, \varphi_2) \in \mathcal{R}\langle\kappa_1 \to \kappa_2\rangle\ t_1\ \ldots\ t_n \\ \equiv\ &\forall x_1\ \ldots\ x_n\,.\,\forall v_1\ v_2\,.\,(v_1, v_2) \in \mathcal{R}\langle\kappa_1\rangle\ x_1\ \ldots\ x_n \\ &\qquad\qquad \supset (\varphi_1\ v_1, \varphi_2\ v_2) \in \mathcal{R}\langle\kappa_2\rangle\ (t_1\ x_1)\ \ldots\ (t_n\ x_n)\ , \end{aligned}$$
3. $\mathcal{R}$ is pointed,
4. $\mathcal{R}$ is chain complete.

Usually, a logical relation will be defined for the base case $\star$; the second clause of the definition then shows how to extend the relation to functional kinds. The third and the fourth condition ensure that a logical relation relates fixpoint operators. It is generally easy to prove that a relation is pointed: note that $\mathcal{R}\langle\kappa_1 \to \kappa_2\rangle$ is pointed if $\mathcal{R}\langle\kappa_2\rangle$ is pointed; hence it suffices to show that $\mathcal{R}\langle\star\rangle$ is pointed. Similarly, $\mathcal{R}\langle\kappa_1 \to \kappa_2\rangle$ is chain complete if $\mathcal{R}\langle\kappa_1\rangle$ and $\mathcal{R}\langle\kappa_2\rangle$ are chain complete; $\mathcal{R}\langle\star\rangle$ is chain complete if it takes the form of an equation.

Given these prerequisites we can now state the Basic Lemma of logical relations.

Lemma 1 (*Basic Lemma*)

Let $\mathcal{R}$ be a logical relation over $Poly_1$ and $Poly_2$ and let $poly_1\langle t\rangle :: Poly_1\langle\kappa\rangle\ t\ \ldots\ t$ and $poly_2\langle t\rangle :: Poly_2\langle\kappa\rangle\ t\ \ldots\ t$ be such that

$$(poly_1\langle c\rangle, poly_2\langle c\rangle) \in \mathcal{R}\langle kind\ c\rangle\ c\ \ldots\ c$$

for every type constant $c \in C$. If η_1 and η_2 are environments and $\theta_1, \ldots, \theta_n$ are type substitutions such that $(\eta_1\ x, \eta_2\ x) \in \mathcal{R}\langle kind\ x\rangle\ (x\theta_1)\ \ldots\ (x\theta_n)$ for every type variable $x \in \mathrm{free}(t)$, then

$$(poly_1\langle t\rangle\ \eta_1, poly_2\langle t\rangle\ \eta_2) \in \mathcal{R}\langle\kappa\rangle\ (t\theta_1)\ \ldots\ (t\theta_n)$$

for every type term $t \in T^\circ(\kappa)$.

Proof

We use induction on the derivation of $u :: \kappa$. **Case** $u = c$: the statement holds trivially since $\mathcal{R}$ relates constants. **Case** $u = x$: the statement holds since η_1 and η_2 are related. **Case** $u = t_1\ t_2$: by the induction hypothesis, we have

$$\begin{aligned} &(poly_1\langle t_1\rangle\ \eta_1, poly_2\langle t_1\rangle\ \eta_2) \in \mathcal{R}\langle\kappa_1 \to \kappa_2\rangle\ (t_1\theta_1)\ \ldots\ (t_1\theta_n) \\ \equiv\ &\forall x_1\ \ldots\ x_n\,.\,\forall v_1\ v_2\,. \\ &\qquad (v_1, v_2) \in \mathcal{R}\langle\kappa_1\rangle\ x_1\ \ldots\ x_n \\ &\qquad\quad \supset ((poly_1\langle t_1\rangle\ \eta_1)\ v_1, (poly_2\langle t_1\rangle\ \eta_2)\ v_2) \in \mathcal{R}\langle\kappa_2\rangle\ ((t_1\theta_1)\ x_1)\ \ldots\ ((t_1\theta_n)\ x_n) \end{aligned}$$

and

$$(poly_1\langle t_2\rangle\ \eta_1, poly_2\langle t_2\rangle\ \eta_2) \in \mathcal{R}\langle\kappa_1\rangle\ (t_2\theta_1)\ \ldots\ (t_2\theta_n)\ ,$$

which implies $(poly_1\langle t_1\ \ t_2\rangle\ \eta_1, poly_2\langle t_1\ \ t_2\rangle\ \eta_2) \in \mathcal{R}\langle\kappa_2\rangle\ ((t_1\ t_2)\theta_1)\ \ldots\ ((t_1\ t_2)\theta_n)$.
Case $u = \Lambda x.t$: we have to show that

$$\begin{aligned}
&(poly_1\langle \Lambda x.t\rangle\ \eta_1, poly_2\langle \Lambda x.t\rangle\ \eta_2) \in \mathcal{R}\langle kind\ x \to \kappa\rangle\ ((\Lambda x.t)\theta_1)\ \ldots\ ((\Lambda x.t)\theta_n)\\
\equiv\quad &\forall x_1\ \ldots\ x_n\,\blacksquare\,\forall v_1\ v_2\,\blacksquare\\
&\qquad (v_1, v_2) \in \mathcal{R}\langle kind\ x\rangle\ x_1\ \ldots\ x_n\\
&\qquad\quad \supset ((poly_1\langle \Lambda x.t\rangle\ \eta_1)\ v_1, (poly_2\langle \Lambda x.t\rangle\ \eta_2)\ v_2)\\
&\qquad\qquad \in \mathcal{R}\langle\kappa\rangle\ (((\Lambda x.t)\theta_1)\ x_1)\ \ldots\ (((\Lambda x.t)\theta_n)\ x_n)\ .
\end{aligned}$$

Assume that $(v_1, v_2) \in \mathcal{R}\langle kind\ x\rangle\ x_1\ \ldots\ x_n$. Since the modified environments $\eta_1(x := v_1)$ and $\eta_2(x := v_2)$ are related, we can invoke the induction hypothesis to obtain

$$\begin{aligned}
&(poly_1\langle t\rangle\ (\eta_1(x := v_1)), poly_2\langle t\rangle\ (\eta_2(x := v_2)))\\
&\quad \in \mathcal{R}\langle\kappa\rangle\ (t(\theta_1(x := x_1)))\ \ldots\ (t(\theta_n(x := x_n)))\ .
\end{aligned}$$

Now, since $((\Lambda x.t)\theta_i)\ x_i = t(\theta_i(x := x_i))$, the statement follows. **Case** $u = \mu_\kappa$: we have to show that

$$\begin{aligned}
&(fix, fix) \in \mathcal{R}\langle(\kappa \to \kappa) \to \kappa\rangle\ \mu\ \ldots\ \mu\\
\equiv\quad &\forall x_1\ \ldots\ x_n\,\blacksquare\,\forall \varphi_1\ \varphi_2\,\blacksquare\,(\varphi_1, \varphi_2) \in \mathcal{R}\langle\kappa \to \kappa\rangle\ x_1\ \ldots\ x_n\\
&\qquad\qquad\qquad\qquad \supset (fix\ \varphi_1, fix\ \varphi_2) \in \mathcal{R}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n)\ .
\end{aligned}$$

Assume that φ_1 and φ_2 are related. By definition of logical relations, we have

$$\begin{aligned}
&(\varphi_1, \varphi_2) \in \mathcal{R}\langle\kappa \to \kappa\rangle\ x_1\ \ldots\ x_n\\
\equiv\quad &\forall y_1\ \ldots\ y_n\,\blacksquare\,\forall v_1\ v_2\,\blacksquare\\
&\qquad (v_1, v_2) \in \mathcal{R}\langle\kappa\rangle\ y_1\ \ldots\ y_n\\
&\qquad\quad \supset (\varphi_1\ v_1, \varphi_2\ v_2) \in \mathcal{R}\langle\kappa\rangle\ (x_1\ y_1)\ \ldots\ (x_n\ y_n)\ .
\end{aligned}$$

Since $\mathcal{R}$ is pointed, we furthermore have $(\bot, \bot) \in \mathcal{R}\langle\kappa\rangle\ 0\ \ldots\ 0$. The polytypic fixpoint induction rule then implies $(fix\ \varphi_1, fix\ \varphi_2) \in \mathcal{R}\langle\kappa\rangle\ (\mu x_1)\ \ldots\ (\mu x_n)$. □

7.3 Examples

7.3.1 Mapping functions

The functorial laws are captured by the logical relations $\mathcal{I}$ and $\mathcal{C}$, which we have discussed at length in Section 7.1.

$$\begin{aligned}
\mathcal{I}\langle\star\rangle\ t\quad &\subseteq\quad t \to t\\
\varphi \in \mathcal{I}\langle\star\rangle\ t\quad &\equiv\quad \varphi = id :: t \to t\\
\mathcal{C}\langle\star\rangle\ t_1\ t_2\ t_3\quad &\subseteq\quad (t_2 \to t_3) \times (t_1 \to t_2) \times (t_1 \to t_3)\\
(\varphi_1, \varphi_2, \varphi_3) \in \mathcal{C}\langle\star\rangle\ t_1\ t_2\ t_3\quad &\equiv\quad \varphi_1 \cdot \varphi_2 = \varphi_3 :: t_1 \to t_3
\end{aligned}$$

It is not hard to see that $\mathcal{I}$ and $\mathcal{C}$ are pointed and chain complete. Since $map\langle c\rangle \in \mathcal{I}\langle kind\ c\rangle\ c$ and $(map\langle c\rangle, map\langle c\rangle, map\langle c\rangle) \in \mathcal{C}\langle kind\ c\rangle\ c\ c\ c$ for all type constants $c \in C$, the Basic Lemma implies $map\langle t\rangle \in \mathcal{I}\langle\kappa\rangle\ t$ and $(map\langle t\rangle, map\langle t\rangle, map\langle t\rangle) \in \mathcal{C}\langle\kappa\rangle\ t\ t\ t$ for all closed type terms $t \in T^\circ(\kappa)$.

7.3.2 Reductions

Using a minor variant of $\mathcal{C}$ we can also relate reductions and mapping functions.

$$\begin{array}{lcl}\mathcal{C}\langle\star\rangle\ t_1\ t_2\ z & \subseteq & (t_2 \to z) \times (t_1 \to t_2) \times (t_1 \to z)\\ (\varphi_1, \varphi_2, \varphi_3) \in \mathcal{C}\langle\star\rangle\ t_1\ t_2\ z & \equiv & \varphi_1 \cdot \varphi_2 = \varphi_3 :: t_1 \to z\end{array}$$

Here, z is some fixed type that is passed unchanged to the base case $\mathcal{C}\langle\star\rangle$. Now, given an element $e :: z$ and an operation $(\oplus) :: z \to z \to z$, we have

$$(\mathit{reduce}\langle t\rangle\ e\ (\oplus), \mathit{map}\langle t\rangle, \mathit{reduce}\langle t\rangle\ e\ (\oplus)) \in \mathcal{C}\langle\kappa\rangle\ t\ t\ z\ .$$

An immediate consequence of this property is

$$\mathit{reduce}\langle f\rangle\ e\ (\oplus)\ \varphi \cdot \mathit{map}\langle f\rangle\ \psi \quad = \quad \mathit{reduce}\langle f\rangle\ e\ (\oplus)\ (\varphi \cdot \psi)\ ,$$

which shows how to fuse a reduction with a map. Now, in order to prove the polytypic property we merely have to verify that the statement holds for every type constant $c \in C$. Using the point-free definitions of *map* and *red* this amounts to showing that

$$\begin{array}{lcl}\mathit{const}\ e \cdot \mathit{id} & = & \mathit{const}\ e\\ (\varphi_1 \triangledown \varphi_2) \cdot (\psi_1 + \psi_2) & = & (\varphi_1 \cdot \psi_1) \triangledown (\varphi_2 \cdot \psi_2)\\ \mathit{uncurry}\ (\oplus) \cdot (\varphi_1 \times \varphi_2) \cdot (\psi_1 \times \psi_2) & = & \mathit{uncurry}\ (\oplus) \cdot ((\varphi_1 \cdot \psi_1) \times (\varphi_2 \cdot \psi_2))\ .\end{array}$$

All three conditions hold.

Previous approaches to polytypic programming (Jansson & Jeuring, 1997; Hinze, 1999) required the programmer to specify the action of a polytypic function for the composition of two type constructors: for instance, for *fsize* the polytypic programmer had to supply the equation $\mathit{fsize}\langle f_1 \cdot f_2\rangle = \mathit{fsum}\langle f_1\rangle \cdot \mathit{map}\langle f_1\rangle\ (\mathit{fsize}\langle f_2\rangle)$. Interestingly, using *reduce-map* fusion this equation can be derived from the definitions of *fsize* and *fsum* given in Fig. 3.

$$\begin{array}{ll} & \mathit{fsize}\langle f_1 \cdot f_2\rangle\\ = & \quad \{\ \text{definition } \mathit{fsize}\ \}\\ & \mathit{reduce}\langle f_1 \cdot f_2\rangle\ 0\ (+)\ (\mathit{const}\ 1)\\ = & \quad \{\ \mathit{poly}\langle f_1 \cdot f_2\rangle = \mathit{poly}\langle f_1\rangle \cdot \mathit{poly}\langle f_2\rangle\ \}\\ & \mathit{reduce}\langle f_1\rangle\ 0\ (+)\ (\mathit{reduce}\langle f_2\rangle\ 0\ (+)\ (\mathit{const}\ 1))\\ = & \quad \{\ \text{definition } \mathit{fsize}\ \}\\ & \mathit{reduce}\langle f_1\rangle\ 0\ (+)\ (\mathit{fsize}\langle f_2\rangle)\\ = & \quad \{\ \mathit{reduce}\text{-}\mathit{map}\ \text{fusion}\ \}\\ & \mathit{reduce}\langle f_1\rangle\ 0\ (+)\ \mathit{id} \cdot \mathit{map}\langle f_1\rangle\ (\mathit{fsize}\langle f_2\rangle)\\ = & \quad \{\ \text{definition } \mathit{fsum}\ \}\\ & \mathit{fsum}\langle f_1\rangle \cdot \mathit{map}\langle f_1\rangle\ (\mathit{fsize}\langle f_2\rangle)\end{array}$$

As a final example let us generalize the fusion law for reductions given by L. Meertens in (Meertens, 1996). To this end we use the logical relation $\mathcal{F}$ defined by

$$\begin{aligned} \mathcal{F}\langle\star\rangle\ t\ z_1\ z_2 &\subseteq (t \to z_1) \times (t \to z_2) \\ (\varphi_1, \varphi_2) \in \mathcal{F}\langle\star\rangle\ t\ z_1\ z_2 &\equiv h \cdot \varphi_1 = \varphi_2 :: t \to z_2\ , \end{aligned}$$

where z_1 and z_2 are fixed types and $h :: z_1 \to z_2$. The polytypic fusion law, which gives conditions for fusing the function h with a reduction, then takes the following form (to ensure that $\mathcal{F}$ is pointed h must be strict)

$$\begin{aligned} & h\ \bot = \bot \\ \cap\quad & h\ n = e \\ \cap\quad & h\ (v \oplus w) = h\ v \otimes h\ w \\ \supset\quad & (\mathit{reduce}\langle t\rangle\ n\ (\oplus), \mathit{reduce}\langle t\rangle\ e\ (\otimes)) \in \mathcal{F}\langle\kappa\rangle\ t\ z_1\ z_2\ . \end{aligned}$$

We can apply this law, for instance, to prove that $\mathit{length} \cdot \mathit{fflatten}\langle f\rangle = \mathit{fsize}\langle f\rangle$.

8 Related work

The idea to assign polykinded types to polytypic values is, to the best of the author's knowledge, original. Previous approaches to polytypic programming (Jansson & Jeuring, 1997; Hinze, 2000) were restricted in that they only allowed to parameterize values by types of one fixed kind. Two notable exceptions are Functorial ML (FML) (Jay *et al.*, 1998) and the work of F. Ruehr (Ruehr, 1992). FML allows to quantify over functor arities in type schemes (since FML handles only regular, first-order functors, kinds can be simplified to arities). However, no formal account of this feature is given and the informal description makes use of an infinitary typing rule. Furthermore, the polytypic definitions based on this extension are rather unwieldy from a notational point of view. F. Ruehr also restricts type indices to types of one fixed kind. Additional flexibility is, however, gained through the use of a more expressive kind language, which incorporates kind variables. This extension is used to define a higher-order map indexed by types of kind $(\alpha \to \star) \to \star$, where α is a kind variable. Clearly, this mapping function is subsumed by the polytypic map given in Section 2. Whether kind polymorphism has other benefits remains to be seen.

9 Conclusion

Haskell possesses a rich type system, which is tantamount to the simply typed lambda calculus (with kinds playing the rôle of types). This type system presents a challenge for polytypism: how can we define polytypic values and how can we assign types to these values? This paper offers satisfactory answers to both questions. It appears that polytypic values possess polykinded types, that is, types that are defined by induction on the structure of kinds. Interestingly, to define a polykinded type it suffices to specify the image of the base kind; likewise, to define a polytypic value it suffices to specify the images of type constants. Everything else comes for free. In fact, the specialization of a polytypic value can be regarded as an interpretation of the simply typed lambda calculus. This renders it possible to adapt one of the main tools for studying typed lambda calculi, logical relations, to polytypic reasoning. To prove a polytypic property it suffices to prove the assertion for type constants. Everything else is taken care of automatically. We have applied this framework to show among other things that the polytypic *map* satisfies polytypic versions of the two functorial laws.

Acknowledgements

I would like to thank Johan Jeuring for stimulating discussions on polytypic programming and for his suggestion to rewrite Section 7. Thanks are furthermore due to Fritz Ruehr for his helpful comments on an earlier draft of this paper.

References

Backhouse, Roland, Jansson, Patrik, Jeuring, Johan, & Meertens, Lambert. (1999). Generic Programming — An Introduction —. *Pages 28–115 of:* Swierstra, S. Doaitse, Henriques, Pedro R., & Oliveira, Jose N. (eds), *3rd international summer school on advanced functional programming, Braga, Portugal*. Lecture Notes in Computer Science, vol. 1608. Berlin: Springer-Verlag.

Bird, Richard, & Meertens, Lambert. (1998). Nested datatypes. *Pages 52–67 of:* Jeuring, J. (ed), *Fourth international conference on mathematics of program construction, MPC'98, Marstrand, Sweden*. Lecture Notes in Computer Science, vol. 1422. Springer-Verlag.

Bird, Richard, & Paterson, Ross. (1999). Generalised folds for nested datatypes. *Formal aspects of computing*, **11**(2), 200–222.

Bird, Richard, de Moor, Oege, & Hoogendijk, Paul. (1996). Generic functional programming with types and relations. *Journal of functional programming*, **6**(1), 1–28.

Danvy, Olivier. (1999). An extensional characterization of lambda-lifting and lambda-dropping. *Pages 241–250 of:* Middeldorp, Aart, & Sato, Taisuke (eds), *4th Fuji international symposium on functional and logic programming (FLOPS'99), Tsukuba, Japan*. Lecture Notes in Computer Science, vol. 1722. Springer-Verlag.

Fokkinga, M.M. 1994 (June). *Monadic maps and folds for arbitrary datatypes*. Tech. rept. Memoranda Informatica 94-28. University of Twente.

Girard, Jean-Yves. (1972). *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieur*. Ph.D. thesis, Université Paris VII.

Henglein, Fritz. (1993). Type inference with polymorphic recursion. *ACM transactions on programming languages and systems*, **15**(2), 253–289.

Hinze, Ralf. (1999). Polytypic functions over nested datatypes. *Discrete mathematics and theoretical computer science*, **3**(4), 159–180.

Hinze, Ralf. 2000 (January). A new approach to generic functional programming. *Pages 119–132 of:* Reps, Thomas W. (ed), *Proceedings of the 27th annual ACM SIGPLAN-SIGACT symposium on principles of programming languages, Boston, Massachusetts, january 19-21*.

Jansson, Patrik, & Jeuring, Johan. (1997). PolyP—a polytypic programming language extension. *Pages 470–482 of: Conference record 24th ACM SIGPLAN-SIGACT symposium on principles of programming languages, POPL'97, Paris, France*. ACM-Press.

Jansson, Patrik, & Jeuring, Johan. (1998). PolyLib—A library of polytypic functions.

Backhouse, Roland, & Sheard, Tim (eds), *Informal proceedings workshop on generic programming, WGP'98, Marstrand, Sweden.* Department of Computing Science, Chalmers University of Technology and Göteborg University.

Jansson, Patrik, & Jeuring, Johan. (1999). *Calculating polytypic data conversion programs.* In submission.

Jay, C.B., Bellè, G., & Moggi, E. (1998). Functorial ML. *Journal of functional programming*, **8**(6), 573–619.

Jones, M.P., & Peterson, J.C. 1999 (May). *Hugs 98 user manual.* Available from `http://www.haskell.org/hugs`.

Leivant, Daniel. (1983). Polymorphic type inference. *Proc. 10th symposium on principles of programming languages.*

McCracken, Nancy Jean. 1979 (June). *An investigation of a programming language with a polymorphic type structure.* Ph.D. thesis, Syracuse University.

Meertens, Lambert. (1996). Calculate polytypically! *Pages 1–16 of:* Kuchen, H., & Swierstra, S.D. (eds), *Proceedings 8th international symposium on programming languages: Implementations, logics, and programs, PLILP'96, Aachen, Germany.* Lecture Notes in Computer Science, vol. 1140. Springer-Verlag.

Meijer, Erik, & Hutton, Graham. (1995). Bananas in space: Extending fold and unfold to exponential types. *Pages 324–333 of: Conference record 7th ACM SIGPLAN/SIGARCH and IFIP WG 2.8 international conference on functional programming languages and computer architecture, FPCA'95, La Jolla, San Diego, CA, USA.* ACM-Press.

Meijer, Erik, & Jeuring, Johan. (1995). Merging monads and folds for functional programming. *Pages 228–266 of:* Jeuring, J., & Meijer, E. (eds), *1st international spring school on advanced functional programming techniques, Båstad, Sweden.* Lecture Notes in Computer Science, vol. 925. Berlin: Springer-Verlag.

Milner, Robin. (1978). A theory of type polymorphism in programming. *Journal of computer and system sciences*, **17**(3), 348–375.

Mitchell, John C. (1996). *Foundations for programming languages.* Cambridge, MA: The MIT Press.

Mycroft, Alan. (1984). Polymorphic type schemes and recursive definitions. *Pages 217–228 of:* Paul, M., & Robinet, B. (eds), *Proceedings of the international symposium on programming, 6th colloquium, Toulouse, France.* Lecture Notes in Computer Science, vol. 167.

Okasaki, Chris. (1998). *Purely functional data structures.* Cambridge University Press.

Peyton Jones, Simon, & Hughes, John (eds). 1999 (February). *Haskell 98 — A non-strict, purely functional language.* Available from `http://www.haskell.org/definition/`.

Peyton Jones, Simon L. (1996). Compiling Haskell by program transformation: A report from the trenches. *Pages 18–44 of:* Nielson, Hanne Riis (ed), *Programming languages and systems—ESOP'96, 6th european symposium on programming, linköping, sweden, 22–24 april.* Lecture Notes in Computer Science, vol. 1058. Springer-Verlag.

Ruehr, Karl Fritz. (1992). *Analytical and structural polymorphism expressed using patterns over types.* Ph.D. thesis, University of Michigan.

Team, The GHC. 1999 (September). *The glasgow haskell compiler user's guide, version 4.04.* Available from `http://www.haskell.org/ghc/documentation.html`.

Wadler, Philip. (1989). Theorems for free! *Pages 347–359 of: The fourth international conference on functional programming languages and computer architecture (FPCA'89), London, UK.* Addison-Wesley Publishing Company.